Facultad
de
Ciencias

RESOLVIENDO PROBLEMAS BIDIMENSIONALES DE EMPAQUETAMIENTO Y CORTE

Solving Two-Dimensional Cutting and Packing Problems

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Miguel Ruiz Granda
Directora: Cecilia Pola Méndez

Septiembre 2021

Índice general

Agradecimientos

Este trabajo supone la finalización de una etapa que empezó hace ya cinco años. El Doble Grado en Física y Matemáticas ha supuesto un gran esfuerzo y dedicación, superiores a las que anticipé cuando me matriculé por primera vez en 2016, pero los conocimientos y competencias adquiridos durante todo el proceso han merecido completamente la pena.

En primer lugar querría dar las gracias a Cecilia, por las innumerables reuniones que hemos tenido a lo largo del último año y la dedicación y el esfuerzo que ha destinado para que este trabajo tuviese la mayor calidad posible.

Por último agradecer el apoyo de mis padres, amigos y mi familia a lo largo de toda mi carrera académica. Habría sido mucho más difícil sin vosotros.

Resumen

En este proyecto se aborda el problema bidimensional de empaquetamiento (2DBPP) que consiste en colocar, sin solapamientos, un conjunto finito de rectángulos en el mínimo número posible de contenedores bidimensionales rectangulares idénticos. Estos problemas tienen numerosas aplicaciones industriales: en el sector de corte (madera, cristal, etc), en el transporte de mercancías, en las telecomunicaciones, etc. El problema es NP-duro y muy difícil de resolver en la práctica en un tiempo razonable. Aún hoy en día la determinación de una buena formulación MIP es un reto. En este trabajo se ha optado por utilizar la formulación y el método de resolución presentado en [10], que es considerado el estado actual del arte en la resolución de (2DBPP) [5]. A lo largo del mismo se han utilizado distintos algoritmos como generación de columnas, MAC, ramificación y poda y heurísticos, junto con métodos de programación lineal continua y entera, alguno de ellos con implementación propia usando los lenguajes de programación Matlab y C. Además, se han realizado experimentos numéricos utilizando la colección de problemas NGCUT [2].

Palabras clave: 2DBPP, generación de columnas, programación entera, pricing problem, set-covering.

Abstract

This project addresses the two-dimensional packaging problem (2DBPP) which consists of allocating, without overlaps, a finite set of rectangles into the minimum number of identical two-dimensional rectangular containers. These problems have many industrial applications: in the cutting area (wood, glass, etc.), in the transportation of goods, in telecommunications, etc. The problem is NP-hard and very difficult to solve in a reasonable time. Even today determining a good MIP formulation is challenging. We use in this work the formulation and resolution method presented in [10], which is considered the current state of the art for solving (2DBPP) [5]. Throughout it, different algorithms have been used such as column generation, MAC, Branch and Bound and heuristics, along with continuous and integer linear programming methods, some of them with our Matlab and C codes. In addition, numerical experiments have been performed on NGCUT problem set [2].

Key words: 2DBPP, column generation, integer programming, pricing problem, set-covering.

Capítulo 1

Introducción

Los problemas bidimensionales de empaquetamiento y corte, conocidos en la literatura científica bajo el nombre de *Two-dimensional Bin Packing Problems* (2DBPP), consisten en colocar, sin solapamientos, un conjunto finito de rectángulos en el mínimo número posible de contenedores rectangulares idénticos. Se asume que los rectángulos no pueden ser rotados y tienen que ser colocados con los bordes paralelos a los del contenedor [6].

Estos problemas tienen un gran número de aplicaciones industriales en ámbitos como el corte (de papel, cristal, madera, etc), la colocación de bienes en almacenes, transporte de mercancías, telecomunicaciones y la paginación de periódicos.

Los problemas (2DBPP) son muy difíciles en general, perteneciendo a la clase de complejidad NP-duro, por lo que no se conoce ningún algoritmo que sea capaz de resolver el problema en tiempo polinomial. Debido a esta situación, en algunos modelos se impone el empaquetamiento por niveles, que si bien puede tener interés en algunos sectores industriales, supone el inconveniente de que no siempre se aprovecha al máximo el espacio del contenedor. El empaquetamiento por niveles da lugar a patrones guillotinables, es decir, aquellos que pueden ser cortados por una cuchilla de lado a lado del contenedor sin partir ningún rectángulo.

En la Figura 1.1 se muestra un ejemplo de un patrón no guillotinable. En el caso de querer colocar los rectángulos por niveles, serían necesarios dos contenedores. En situaciones en las que no se presentan restricciones tecnológicas para el corte y minimizar el desperdicio es uno de los objetivos a lograr, resulta muy beneficioso no utilizar empaquetamiento por niveles.

El modelo y el procedimiento de resolución utilizados en este proyecto están basados en [10], donde se usa un método de generación de columnas para seleccionar un subconjunto de posibles empaquetamientos de un contenedor y no se imponen cortes guillotinables, garantizando un buen aprovechamiento de los contenedores utilizados. Otro aspecto crítico para la elección de esta referencia es que desde el punto de vista computacional es considerada actualmente el estado del arte para la resolución de problemas bidimensionales de empaquetamiento [5].

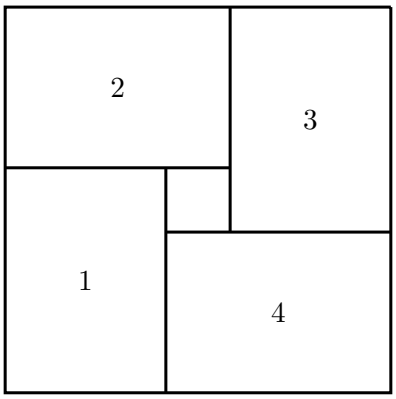


Figura 1.1: Patrón de rectángulos que no es guillotinable.

Esta memoria se encuentra estructurada de la siguiente forma. En primer lugar, en el capítulo 2 se introduce la formulación del problema así como la conversión a forma matricial del mismo para que pueda ser resuelto por un resolvedor númerico especializado en problemas MIP. En segundo lugar, en el capítulo 3 se presenta una reformulación del problema del capítulo 2 en términos de un problema de cobertura de conjuntos y se introduce el método de resolución del problema basado en la generación de columnas. Esta técnica permite abordar la resolución de problemas que por su gran talla son inviables de resolver directamente, tratando otros problemas con un número de variables menor y que va aumentando a lo largo del proceso. Como rasgo general, el método permite descomponer el problema en dos: el Restricted Master Problem (RMP) y el Pricing Problem (PP). En el capítulo 4 se explican los algoritmos que se utilizan para resolver el Pricing Problem, así como las distintas cotas inferiores utilizadas en este proyecto sobre el número necesario de contenedores para realizar el empaquetamiento. Por último, en el capítulo 5 se muestran los resultados numéricos realizados con el conjunto de 12 problemas de la colección NGCUT obtenidos de [9], siendo 22 el mayor cardinal del conjunto de rectángulos a empaquetar (notemos que en [10] se resuelven problemas de hasta 100 rectángulos).

Finalmente, apreciemos que la parte de implementación del método ha supuesto un gran esfuerzo por requerir el abordaje de distintos métodos para resolver los distintos problemas que aparecen: CSP, LP, MIP e IP. Se han usado programas propios en Matlab así como otros incluidos en dicho paquete (`linprog` e `intlinprog`) y de otros autores como el programa MAC [7] (en este caso ha sido necesario hacer una interfaz entre C y Matlab).

Capítulo 2

Formulación del problema

La dificultad de los problemas (2DBPP) es muy elevada en general, es por ello que se va a realizar algún tipo de simplificación en la formulación del problema como la de considerar únicamente objetos y contenedores rectangulares, así como todos los contenedores del mismo tamaño. El modelo escogido es el que se aparece en [10].

2.1. Formulación general del modelo

Se parte de un conjunto de rectángulos R identificados con los elementos del conjunto de índices $I_R = \{1, \ldots, n\}$. Para cada rectángulo i del conjunto R se conoce su altura, h_i, y su anchura, w_i. Además, para colocar todos los objetos se dispone de un número suficiente (al menos n) de contenedores rectangulares ordenados, todos con la misma altura, H, y anchura, W.

En la formulación del problema aparecen distintos tipos de variables. Unas de las más importantes son aquellas relacionadas con la posición espacial de cada rectángulo del conjunto R. Para ello, para cada rectángulo i se dispone de las coordenadas de la esquina inferior izquierda (x_i, y_i) y del número de contenedor al que pertenece, m_i. La más destacada es la que indica el número de contenedores utilizados, v, que a su vez actúa de cota superior de cada variable m_i.

A partir de las variables ya explicadas, se definen un conjunto de variables auxiliares. Entre ellas, encontramos las variables binarias, l_{ij}, b_{ij} y p_{ij}, que hacen referencia a la posición relativa de dos rectángulos distintos i y j del conjunto R. Si la variable l_{ij} (*left*) toma el valor 1 entonces el rectángulo i está a la izquierda del rectángulo j, si la variable b_{ij} (*below*) toma el valor 1 entonces el rectángulo i está debajo del rectángulo j y si la variable p_{ij} (*previous*) toma el valor 1 entonces el rectángulo i pertenece a un contenedor con un identificador menor que el contenedor al que pertenece el rectángulo j.

En último lugar, aparecen las restricciones del problema, que son condiciones en las que se relacionan las distintas variables descritas previamente. Las restricciones principales que se encuentran son las asociadas al tamaño, al número de contenedor y al solapamiento entre rectángulos.

Las restricciones asociadas al tamaño tienen en cuenta que al colocar una pieza esta

no se salga del contenedor. Además, se tienen las restricciones relacionadas con el número de contenedor, de forma que el número total de contenedores utilizados sea mayor que el índice de cualquier contenedor usado. Estas restricciones se encuentran formuladas matemáticamente en las siguientes restricciones.

$$0 \leq x_i \leq W - w_i, \quad 0 \leq y_i \leq H - h_i. \tag{2.1}$$

$$1 \leq m_i \leq v. \tag{2.2}$$

Nótese que previamente al definir el conjunto de datos no se había puesto ningún tipo de restricción, es decir, era posible tomar un tamaño de contenedor menor que alguno de los rectángulos. A partir de las restricciones de (2.1) se comprueba trivialmente que en ese caso no existe solución al problema ya que se violan las restricciones.

Las variables binarias l_{ij}, b_{ij} y p_{ij} permiten disponer de variables indicadoras sobre la posición relativa entre dos rectángulos, lo cual es fundamental para establecer condiciones que permitan saber si dos rectángulos se solapan o no. En el lenguaje coloquial se puede decir que dos objetos no están superpuestos o no se solapan cuando es posible establecer algún tipo de relación entre ambos, por ejemplo, cuando un objeto está por debajo de otro, o cuando está a su izquierda. Traducido al lenguaje matemático, esta condición o restricción de no solapamiento "natural" para dos rectángulos i y j del conjunto R que verifican que $i < j$ viene dada por la siguiente desigualdad:

$$l_{ij} + l_{ji} + b_{ij} + b_{ji} + p_{ij} + p_{ji} \geq 1. \tag{2.3}$$

Las desigualdad anterior introduce la primera restricción asociada al solapamiento. Veamos otras que la complementan. Estamos interesados en introducir dicha información como restricciones lineales, ya que es conocido que un problema de optimización es más fácil de tratar cuando las restricciones son lineales que cuando no lo son.

Sean dos rectángulos i y j verificando que $i < j$ y que no se solapan, por lo que se verifica la restricción (2.3). Consecuentemente al menos uno de los términos debe valer uno. Para el siguiente desarrollo se asume que los términos ij son los que pueden valer uno, ya el razonamiento es el mismo para los términos ji. Por lo tanto, bajo la asunción previa puede ocurrir alguna de las siguientes tres posibilidades:

$$l_{ij} = 1 \Rightarrow x_i + w_i \leq x_j \Leftrightarrow x_i - x_j + W \leq W - w_i. \tag{2.4}$$

$$b_{ij} = 1 \Rightarrow y_i + h_i \leq y_j \Leftrightarrow y_i - y_j + H \leq H - h_i. \tag{2.5}$$

$$p_{ij} = 1 \Rightarrow m_i + 1 \leq m_j \Leftrightarrow m_i - m_j + n \leq n - 1. \tag{2.6}$$

¿Y qué ocurre cuando las variables binarias l_{ij}, b_{ij} o p_{ij} valen cero? Pues bien, en ese caso haciendo uso de las restricciones de (2.1) y (2.2) y el hecho de que $v \leq n$, se obtienen las siguientes relaciones para cualesquiera rectángulos i y j del conjunto R:

$$x_i - x_j \leq W - w_i. \tag{2.7}$$

$$y_i - y_j \leq H - h_i. \tag{2.8}$$

$$m_i - m_j \leq n - 1. \tag{2.9}$$

Combinando dos a dos las ecuaciones (2.4)−(2.9) se llega fácilmente a tres restricciones válidas para cualquier par de rectángulos i y j pertenecientes al conjunto R. Dichas restricciones se muestran a continuación.

$$x_i - x_j + Wl_{ij} \leq W - w_i. \tag{2.10}$$

$$y_i - y_j + Hb_{ij} \leq H - h_i. \tag{2.11}$$

$$m_i - m_j + np_{ij} \leq n - 1. \tag{2.12}$$

Nótese que las tres restricciones obtenidas son lineales, por lo tanto el objetivo buscado ha sido conseguido. Combinando la ecuación (2.3) con las ecuaciones (2.10), (2.11) y (2.12) se han obtenido todas las restricciones de no solapamiento entre dos rectángulos cualesquiera del conjunto R.

Es interesante en este punto realizar un análisis de las variables y restricciones introducidas para comprobar que no se generan incompatibilidades. Para ello se enuncian y demuestran dos proposiciones auxiliares.

Proposición 2.1. *Las variables binarias l_{ij}, b_{ij} y p_{ij} verifican las siguientes propiedades:*

(a) Si $l_{ij} = 1$, entonces $l_{ji} = 0$.
(b) Si $b_{ij} = 1$, entonces $b_{ji} = 0$.
(c) Si $p_{ij} = 1$, entonces $p_{ji} = 0$.

Demostración. Basta probarlo para uno de los tres casos por la similitud de las restricciones (2.10), (2.11) y (2.12). Por ello se va a usar la reducción al absurdo para obtener la propiedad (a). Partiendo de la ecuación (2.10) y suponiendo $l_{ij} = 1$, se tiene que $x_i + w_i \leq x_j$. Si además se tuviera $l_{ji} = 1$, se tendría $x_j + w_j \leq x_i$. Esto da lugar a la siguiente cadena de desigualdades:

$$x_j + w_j \leq x_i \leq x_i + w_i \leq x_j.$$

Esto implica que $w_j \leq 0$, llegando a contradicción con el hecho de que las anchuras de los rectángulos son siempre positivas. □

Proposición 2.2. *Si $m_i = m_j$, entonces $p_{ij} = p_{ji} = 0$ y $2 \geq l_{ij} + l_{ji} + b_{ij} + b_{ji} \geq 1$.*

Demostración. Utilizando (2.12) y la hipótesis $m_i - m_j = 0$, tenemos que

$$\begin{aligned} np_{ij} &\leq n - 1, \\ np_{ji} &\leq n - 1. \end{aligned} \tag{2.13}$$

Usando las dos ecuaciones anteriores y que las variables p_{ij} y p_{ji} son binarias, se deduce que $p_{ij} = p_{ji} = 0$. Utilizando ahora la desigualdad (2.3), se obtiene que $l_{ij}+l_{ji}+b_{ij}+b_{ji} \geq 1$. Finalmente, haciendo uso de las propiedades demostradas en la Proposición 2.1 se deduce que $l_{ij} + l_{ji} \leq 1$ y $b_{ij} + b_{ji} \leq 1$ y, por tanto, que $l_{ij} + l_{ji} + b_{ij} + b_{ji} \leq 2$. □

Por último, con el fin de romper algunas de las simetrías existentes, se añade una restricción adicional para cada rectángulo i del conjunto R que se puede ver en la siguiente desigualdad.

$$m_i \leq i. \tag{2.14}$$

El objetivo es seleccionar los contenedores en los que puede ir cada rectángulo, de forma que si i es el índice que le corresponde, ese rectángulo está colocado en uno de los primeros i contenedores.

Teniendo en cuenta todas las variables descritas previamente y las restricciones que aparecen en (2.1), (2.2), (2.3), (2.10)−(2.12) y (2.14) se obtiene un problema de programación entera mixta, en sus siglas en inglés MIP (*Mixed Integer Programming*), donde minimizamos el número de contenedores necesarios para empaquetar todos los rectángulos:

$$\text{(2DBPP)} \begin{cases} \text{Minimizar } v \\ l_{ij}, b_{ij}, p_{ij} \in \{0,1\}, \quad \forall i,j \in I_R, i \neq j \\ x_i, y_i \in \mathbb{R}, \quad \forall i \in I_R \\ m_i, v \in \mathbb{N}, \quad \forall i \in I_R \\ \text{Sujeto a:} \\ \quad l_{ij} + l_{ji} + b_{ij} + b_{ji} + p_{ij} + p_{ji} \geq 1 \quad \forall i,j \in I_R, i < j \\ \quad x_i - x_j + W l_{ij} \leq W - w_i \quad \forall i,j \in I_R, i \neq j \\ \quad y_i - y_j + H b_{ij} \leq H - h_i \quad \forall i,j \in I_R, i \neq j \\ \quad m_i - m_j + n p_{ij} \leq n - 1 \quad \forall i,j \in I_R, i \neq j \\ \quad 0 \leq x_i \leq W - w_i \quad \forall i \in I_R \\ \quad 0 \leq y_i \leq H - h_i \quad \forall i \in I_R \\ \quad 1 \leq m_i \leq v \quad \forall i \in I_R \\ \quad m_i \leq i \quad \forall i \in I_R. \end{cases} \tag{2.15}$$

Para valorar la complejidad de la formulación del problema escogida es interesante calcular el número de variables y de restricciones de (2.15):

- **Número de variables**: $3(n^2 - n)$ binarias ($l_{ij}, b_{ij}, p_{ij} \in \{0,1\}, \quad \forall i,j \in R, i \neq j$), $2n$ continuas ($x_i, y_i \in \mathbb{R}, \quad \forall i \in R$) y $n+1$ enteras ($m_i, v \in \mathbb{N}, \quad \forall i \in R$). El número total de variables es de $3n^2 + 1$, lo que supone un número de variables que escala de forma cuadrática ($O(n^2)$ variables) con el número de rectángulos, n.
- **Número de restricciones**: se procede a realizar un cálculo detallado de todas ellas.

  1. Las dadas en (2.3) constituyen con un total de $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$ restricciones.
  2. Las de (2.10)−(2.12) contribuyen al total con $3(n^2 - n)$ restricciones.
  3. Las de (2.1), (2.2) y (2.14) contribuyen al total con $7n$ restricciones.

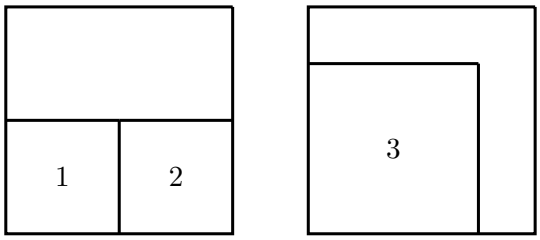
  El número total de restricciones es de $\frac{7}{2}(n^2 + n)$, lo que supone un incremento que escala de forma cuadrática ($O(n^2)$) con el número de rectángulos, n.

El modelo escogido para resolver el problema es complejo ya que tanto las variables como las restricciones crecen de forma cuadrática con el cardinal del conjunto R. Además, la existencia de variables de distinta naturaleza (enteras, binarias y continuas) hace que el problema sea muy exigente desde el punto de vista computacional.

Ejemplo 2.3. *Sea* $I_R = \{1, 2, 3\}$ *tal que* $(w_1, h_1) = (w_2, h_2) = (2, 2)$ *y* $(w_3, h_3) = (3, 3)$. *Se pretende colocar los objetos en un máximo de dos contenedores cuadrados de dimensiones* $W = H = 4$ *¿Cuáles son algunas de las posibles soluciones y valores asociados de las distintas variables de acuerdo al modelo* (2.15)*?*

Mediante este ejemplo práctico sencillo, se pretende ayudar a entender mejor el modelo expuesto. Para este ejemplo se presentan solamente tres posibles soluciones. El cuadrado 3 requiere un contenedor para el solo, por lo que se requiere mínimo dos contenedores. Aquí se presentan algunas de las infinitas soluciones del problema. Resaltar nuevamente que las variables m_i, x_i e y_i son las importantes y las que determinan completamente la solución. Nótese que en el caso de la **Solución 3**, esta podría estar asociada a tres pares de valores (l_{12}, b_{12}): $(1, 0)$, $(0, 1)$ y $(1, 1)$, siendo este último par el que aparece a la derecha del dibujo. Que una misma distribución de rectángulos pueda ser parametrizada de tres formas distintas puede parecer un problema, pero nada más lejos de la realidad. En este trabajo estamos interesados exclusivamente en que las posiciones de los rectángulos estén bien determinadas. Las variables auxiliares han sido creadas con el único objetivo de establecer las condiciones de solapamiento y, por tanto, cualquier problema de unicidad en la descripción de la solución relativo a estas variables es irrelevante en este proyecto.

- **Solución 1**

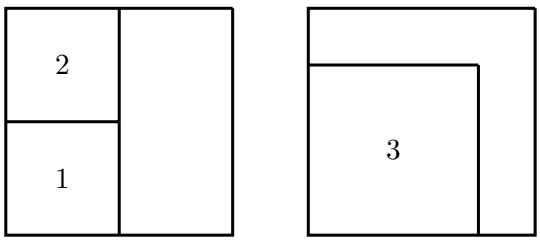


$(x_1, y_1) = (x_3, y_3) = (0, 0)$
$(x_2, y_2) = (2, 0)$
$m_1 = m_2 = 1,\ m_3 = 2$

$l_{12} = 1,\ p_{13} = p_{23} = 1$
El resto de variables binarias son 0.

Figura 2.1: Solución 1. Una de las infinitas soluciones posibles.

- **Solución 2**

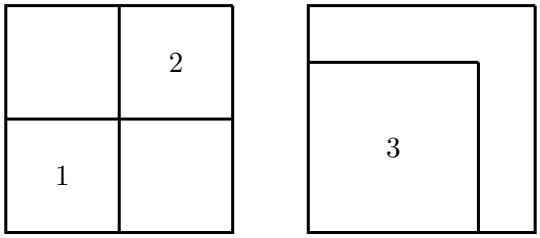


$(x_1, y_1) = (x_3, y_3) = (0, 0)$
$(x_2, y_2) = (0, 2)$
$m_1 = m_2 = 1,\ m_3 = 2$

$b_{12} = 1,\ p_{13} = p_{23} = 1$
El resto de variables binarias son 0.

Figura 2.2: Solución 2. Una de las infinitas soluciones posibles.

- **Solución 3**



$(x_1, y_1) = (x_3, y_3) = (0, 0)$
$(x_2, y_2) = (2, 2)$
$m_1 = m_2 = 1,\ m_3 = 2$

$l_{12} = 1,\ b_{12} = 1$
$p_{13} = p_{23} = 1$

Figura 2.3: Solución 3. Una de las infinitas soluciones posibles.

2.2. Formulación matricial del modelo

La formulación presentada en (2.15) tiene la ventaja de que es muy visual y tanto las restricciones como la función objetivo son fácilmente identificables. El objetivo final del trabajo es ser capaz de resolver casos prácticos mediante el uso del cálculo numérico, por ello, es necesario convertir a matrices las restricciones generales que aparecen en (2.15). Para ello, se busca traducir el (2DBPP) a la notación (MILP), cuyas siglas corresponden a *Mixed Integer Linear Programming* ya que aparecen involucradas variables de distinto tipo. Dicha notación es utilizada por distintos programas de programación lineal entera como, por ejemplo, el software `intlinprog` de Matlab y que se muestra a continuación.

$$(\text{MILP}) \begin{cases} \underset{\hat{x}}{\text{Minimizar}} & c^T \hat{x} \\ \text{Sujeto a:} & A\hat{x} \leq b \\ & l_b \leq \hat{x} \leq u_b \end{cases} \tag{2.16}$$

Para convertir el modelo que aparece en (2.15) al formato matricial de (2.16) se considera el siguiente vector con todas las variables del problema:

$$\hat{x} = (\{l_{ij}\}_{\substack{i,j\in I_R \\ i\neq j}}, \{b_{ij}\}_{\substack{i,j\in I_R \\ i\neq j}}, \{p_{ij}\}_{\substack{i,j\in I_R \\ i\neq j}}, \{x_i\}_{i\in I_R}, \{y_i\}_{i\in I_R}, \{m_i\}_{i\in I_R}, v)^T.$$

El orden escogido para las variables binarias que aparecen en el vector $\hat{x}$ es el que se muestra a continuación. Se ha escrito el orden solamente para el caso de las variables binarias l_{ij} ya que para las variables binarias b_{ij} y p_{ij} es exactamente igual.

$$\begin{aligned} l_{12}, l_{13}, \ldots, l_{1n}, l_{21}, l_{31}, \ldots, l_{n1} &\quad \text{variables en las que aparece 1 como subíndice,} \\ l_{23}, \ldots, l_{2n}, l_{32}, \ldots, l_{n2} &\quad \text{variables en las que aparece 2 como subíndice,} \\ \ldots\ldots & \\ l_{n-1n}, l_{nn-1} &\quad \text{variables en las que aparece } n-1 \text{ como subíndice.} \end{aligned} \tag{2.17}$$

Las filas de variables mostradas arriba con estructura piramidal invertida son concatenadas dando lugar al vector de variables $\{l_{ij}\}_{i,j\in R,\ i\neq j}$. Aunque a priori parezca un orden poco natural y quizás hubiera parecido más razonable utilizar el orden lexicográfico en $\mathbb{N}^2$, con el orden escogido se llega a una forma más sencilla de la matriz de restricciones A.

El vector de la función objetivo viene dado por:

$c^T = (0, \ldots, 0, 1) \in \mathbb{N}^{3n^2+1}$.

El vector de los términos independientes de las restricciones generales viene dado por:

$b = (\overbrace{-1, \ldots, -1}^{\frac{n^2-n}{2}}, K, Q, \overbrace{n-1, \ldots, n-1}^{n^2-n}, \overbrace{0, \ldots, 0}^{n})^T$ donde los vectores $K, Q \in \mathbb{R}^{n^2-n}$ y están definidos como:

$K = (K_1, K_2, \ldots, K_{n-1}, K_n)$ siendo $K_i = (W - w_i, \ldots, W - w_i) \in \mathbb{R}^{n-1}$, $\forall i \in R$.

$Q = (Q_1, Q_2, \ldots, Q_{n-1}, Q_n)$ siendo $Q_i = (H - h_i, \ldots, H - h_i) \in \mathbb{R}^{n-1}$, $\forall i \in R$.

Los vectores de cota inferior y superior vienen dados respectivamente por:

$l_b = (\overbrace{0, \ldots, 0}^{3n^2-n}, \overbrace{1, \ldots, 1}^{n+1})$.

$u_b = (\overbrace{1, \ldots, 1}^{3(n^2-n)}, W - w_1, \ldots, W - w_n, H - h_1, \ldots, H - h_n, 1, \ldots, n, n)$.

La matriz de restricciones A viene dada por:

$$A = \left(\begin{array}{c|c|c|c|c|c|c} M_1 & M_1 & M_1 & & & & \\ \hline W \cdot I_{n^2-n} & & & M_2 & & & \\ \hline & H \cdot I_{n^2-n} & & & M_2 & & \\ \hline & & n \cdot I_{n^2-n} & & & M_2 & \\ \hline & & & & & I_n & e \end{array}\right)$$

donde:

- Las filas de las tres matrices $M_1 \in \mathbb{N}^{\frac{n^2-n}{2}, n^2-n}$ están asociadas a las restricciones (2.3) de forma que las variables binarias se encuentran en el orden mostrado en (2.17).

Sea $\widetilde{I}_i = (-I_i \ -I_i)_{i\times 2i}$, entonces la matriz M_1 estaría descrita por la expresión que aparece en (2.18).

$$M_1 = \begin{pmatrix} \widetilde{I}_{n-1} & & & \\ & \widetilde{I}_{n-2} & & \\ & & \ddots & \\ & & & \widetilde{I}_1 \end{pmatrix} \tag{2.18}$$

- Las submatrices $W \cdot I_{n^2-n}$, $H \cdot I_{n^2-n}$, $n \cdot I_{n^2-n} \in \mathbb{R}^{n^2-n,n^2-n}$ junto con $M_2 \in \mathbb{N}^{n^2-n,n}$ tienen en cuenta las restricciones asociadas a (2.10), (2.11) y (2.12) respectivamente. Para describir la apariencia de la matriz M_2, se va a realizar una descomposición de dicha matriz en bloques en (2.19).

$$M_2 = \begin{pmatrix} \widetilde{M}_{n-1} \\ \widetilde{M}_{n-2} \\ \vdots \\ \widetilde{M}_1 \\ \widetilde{M}_0 \end{pmatrix} \tag{2.19}$$

donde:

$$\widetilde{M}_{n-1} = \left(\begin{array}{c|c} 1 & \\ \vdots & -I_{n-1} \\ 1 & \end{array}\right) \qquad \widetilde{M}_0 = \left(\begin{array}{c|c} & 1 \\ -I_{n-1} & \vdots \\ & 1 \end{array}\right)$$

$$\widetilde{M}_k = \left(\begin{array}{c|c|c} & 1 & \\ -I_{n-k-1} & \vdots & 0_{n-k-1,k} \\ & 1 & \\ \hline & 1 & \\ 0_{k,n-k-1} & \vdots & -I_k \\ & 1 & \end{array}\right) \qquad \forall k \in \{1, 2, \ldots, n-2\}$$

- La matriz I_n junto con el vector $e = (-1, \ldots, -1)^T \in \mathbb{N}^n$ tienen en cuenta la restricción general encapsulada dentro de (2.2).
- Las cajas que se han dejado en blanco corresponden a matrices nulas de distintas dimensiones.

En este punto es importante pararse a reflexionar sobre la matriz A obtenida. En primer lugar, las dimensiones de la matriz A serían $(\frac{7}{2}n^2 - \frac{5}{2}n) \times (3n^2 + 1)$. A modo de ilustración, para un número de rectángulos que se podría considerar bajo, por ejemplo, $n = 20$, la matriz A tendría dimensiones 1350×1201: es enorme. Debido a que el objetivo es utilizar recursos computacionales para resolver numéricamente el problema, se necesitará almacenar esa matriz en la memoria del ordenador. El problema está en que la memoria del ordenador es limitada y la matriz A escala demasiado rápido, lo cual puede implicar que

no se disponga de suficiente memoria para almacenarla a partir de un número determinado de rectángulos.

El aspecto clave de la matriz A a tener en cuenta para resolver este problema es el hecho de que está muy hueca (gran porcentaje de elementos iguales a cero). La solución es ahora evidente: hay que almacenar solamente los valores que no son nulos. Existe un programa en Matlab llamado `sparse` que permite ahorrar mucha memoria en el almacenamiento de matrices utilizando dicha técnica.

Los resultados numéricos presentados en el capítulo 5 muestran que la utilización de `intlinprog` directamente para la resolución de este problema no es muy aconsejable. En muchos de los problemas que se han intentado resolver se alcanzó el límite de tiempo establecido sin haber llegado a una solución. Tal como mencionan en [10], el modelo presentado en (2.15) es demasiado complejo para ser abordado directamente por un resolvedor general de problemas de programación entera mixta como `intlinprog`. Los malos resultados obtenidos con esta herramienta y utilizando esta aproximación al problema motivan la búsqueda y desarrollo de una nueva formulación del problema presentada en el Capítulo 3.

Capítulo 3

Otro enfoque: modelo set-covering

En este capítulo se pretende resolver el modelo (2DBPP) explicado en el capítulo 2. Para conseguirlo se va a abordar el problema utilizando un enfoque distinto al capítulo anterior, tratándolo como un problema de optimización de cobertura de conjuntos o *set-covering* y utilizando un método de generación de columnas para su resolución.

3.1. Modelo set-covering

Se basa en la enumeración de todos los posibles patrones o empaquetamientos (subconjuntos de I_R) para un contenedor. Para ello, se ha optado por emplear la notación matricial que aparece en [6] en la que cada patrón es una columna de una matriz S. Dicha matriz tiene dimensiones $n \times m$, siendo n el número total de rectángulos a empaquetar y m el número total de patrones posibles y verifica que:

$$s_{ij} = \begin{cases} 1 \text{ , si el objeto } i \text{ pertenece al patrón } j. \\ 0 \text{ , en caso contrario.} \end{cases}$$

Otra formulación para el problema (2DBPP) que aparece en (2.15) es en términos de un problema de optimización de cobertura de conjuntos (SCM), *Set-Covering Model* en inglés, como se muestra a continuación.

$$(\text{SCM}) \begin{cases} \text{Minimizar} & \displaystyle\sum_{j=1}^{m} x_j \\ x \in \{0,1\}^m & \\ \text{Sujeto a:} & \displaystyle\sum_{j=1}^{m} s_{ij} x_j \geq 1, \ i = 1, \ldots, n. \end{cases} \tag{3.1}$$

En (SCM) se está minimizando el número de contenedores (patrones de empaquetamiento) requeridos de forma que cada rectángulo sea incluido en algún contenedor. Nótese que debido a la restricción utilizada puede suceder que en una solución óptima un mismo

rectángulo, es decir, asociado a un único índice de I_R, aparezca en dos contenedores distintos. Esto no supone ningún tipo de problema en la práctica ya que lo que realmente importa es el número mínimo de contenedores requeridos.

Una forma de obtener una cota inferior razonablemente buena para el óptimo de la función objetivo del problema de programación entera (SCM) es realizar una relajación continua LP. Esto consiste en relajar la condición de variables binarias y considerarlas reales en el intervalo $[0, +\infty)$. El problema relajado recibe el nombre de (MP), cuyas siglas corresponden a *Master Problem* en inglés, y se muestra en (3.2).

$$(\text{MP}) \begin{cases} \text{Minimizar } \displaystyle\sum_{j=1}^{m} x_j \\ x_j \geq 0,\ j = 1, \ldots, m \\ \text{Sujeto a: } \displaystyle\sum_{j=1}^{m} s_{ij} x_j \geq 1,\ i = 1, \ldots, n. \end{cases} \tag{3.2}$$

En este proyecto se han considerado además otras relajaciones como usar las restricciones $1 \geq x_j \geq 0$, pero al observar que la resolución numérica nos planteaba muchas dificultades técnicas hemos optado por la relajación de [10].

Debido a que el valor de m puede ser muy elevado, se va a optar por utilizar un método de generación de columnas para su resolución.

3.2. Método de generación de columnas

Este método consiste en resolver iterativamente el problema (MP). Como punto de partida, se toma un subconjunto de patrones o lo que es lo mismo un subconjunto de columnas de la matriz S tal que existe un punto admisible del problema. De esta forma, en cada iteración se tiene un subconjunto de m' patrones fijado que da lugar a un *Restricted Master Problem*, (RMP), que se muestra en (3.3). Sin pérdida de generalidad y con el fin de aligerar la notación, en (3.3) se está asumiendo que el subconjunto de patrones tomados coinciden con las m' primeras columnas de la matriz S.

$$(\text{RMP}) \begin{cases} \text{Minimizar } \displaystyle\sum_{j=1}^{m'} x_j \\ x_j \geq 0,\ j = 1, \ldots, m' \\ \text{Sujeto a: } \displaystyle\sum_{j=1}^{m'} s_{ij} x_j \geq 1,\ i = 1, \ldots, n. \end{cases} \tag{3.3}$$

Una vez resuelto este problema reducido se trata de determinar si la solución obtenida lo es también del (MP) tomando como cero las variables asociadas a patrones no considerados en el (RMP). En caso negativo, habrá que añadir más patrones al problema restringido. Para tomar esa decisión recurrimos a las condiciones de optimalidad de primer orden (Teorema A.5).

Para decidir qué patrón (columna de S) añadir se resuelve el llamado *Pricing Problem*, (PP), donde se trata de determinar la columna de S, $\bar{u} = (u_1, u_2, \ldots, u_n)^T$, con el menor coste reducido asociado $c_{Red} = 1 - \bar{u}^T \hat{\mu}$, siendo $\hat{\mu}$ un vector columna de los multiplicadores de Lagrange asociados a las restricciones lineales de desigualdad del (RMP). El *Pricing Problem* es un problema de la mochila bidimensional (*2-Dimensional Knapsack Problem* en inglés), (2DKP), con respecto a los multiplicadores de Lagrange $\hat{\mu}_i$, las dimensiones de los rectángulos (w_i, h_i) y el tamaño de la mochila, en este caso correspondiente a las dimensiones del contenedor, (W, H). El (2DKP) puede ser modelado utilizando además de las variables $\{l_{ij}\}_{i,j \in I_R,\ i \neq j}$, $\{b_{ij}\}_{i,j \in I_R,\ i \neq j}$ y $\{(x_i, y_i)\}_{i \in I_R}$, las variables binarias $\{u_i\}_{i \in I_R}$:

$$u_i = \begin{cases} 1 \text{ , si el rectángulo } i \text{ está empaquetado en el contenedor.} \\ 0 \text{ , en caso contrario.} \end{cases} \tag{3.4}$$

El *Pricing Problem* puede ser formulado ahora como un problema de programación lineal mixto:

$$\text{(PP)} \begin{cases} \text{Maximizar } \displaystyle\sum_{i=1}^{n} \hat{\mu}_i u_i \\ l_{ij}, b_{ij} \in \{0,1\}, \quad \forall i, j \in I_R, i \neq j \\ u_i \in \{0,1\}, \quad \forall i \in I_R \\ x_i, y_i \in \mathbb{R}, \quad \forall i \in I_R \\ \text{Sujeto a:} \\ \qquad l_{ij} + l_{ji} + b_{ij} + b_{ji} + (1 - u_i) + (1 - u_j) \geq 1, \quad \forall i, j \in I_R, i < j \\ \qquad x_i - x_j + W l_{ij} \leq W - w_i, \quad \forall i, j \in I_R, i \neq j \\ \qquad y_i - y_j + H b_{ij} \leq H - h_i, \quad \forall i, j \in I_R, i \neq j \\ \qquad 0 \leq x_i \leq W - w_i, \quad \forall i \in I_R \\ \qquad 0 \leq y_i \leq H - h_i, \quad \forall i \in I_R. \end{cases} \tag{3.5}$$

Se puede observar que en la formulación de (PP) aparecen algunas de las restricciones de (2.15). Dicho esto, resulta importante destacar el papel de las nuevas variables u_i introducidas en (PP). Si $u_i = u_j = 1$ para dos distintos rectángulos i y j de R, entonces ambos rectángulos son empaquetados en el mismo contenedor. En ese caso las primeras restricciones aseguran que esos dos rectángulos no se solapen. Por el contrario, si alguna de las variables u_i o u_j es igual a 0 entonces las primeras restricciones no afectan, ya que en ese caso, a efectos del problema el solapamiento o no de dichos rectángulos es irrelevante.

En este momento la gran cuestión es ¿cuál es la relación entre la resolución del del (MP) y del (RMP)? Esta pregunta la responde el teorema 3.1.

Teorema 3.1. *Sea $\bar{x} \in \mathbb{R}^{m'}$ solución de (RMP) con $\hat{\mu}$ vector de multiplicadores de Lagrange asociado a las restricciones lineales generales y $\bar{u}$ solución del (PP). Si $1 - \bar{u}^T \hat{\mu} \geq 0$, entonces $\bar{y} = (\bar{x}^T, 0^T) \in \mathbb{R}^m$ es solución global de (MP).*

Demostración. Sea el *Master Problem* (MP) y el *Restricted Master Problem* (RMP) definidos en (3.2) y (3.3) respectivamente. Para demostrar este teorema hay que aplicar las condiciones de optimalidad necesarias y suficientes (Teorema A.5) a ambos problemas.

Aplicando primero las condiciones necesarias del teorema A.5 al problema (RMP), la ecuación (A.3) permite obtener las m' ecuaciones siguientes:

$$1 - \sum_{j=1}^{n} \hat{\mu}_j s_{ji} - \underline{\mu_i} = 0, \quad i = 1, \ldots, m' \tag{3.6}$$

donde $\hat{\mu}$ y $\underline{\mu}$ son los multiplicadores de Lagrange del problema (RMP) asociados a las restricciones generales y de cota inferior, respectivamente. Nuevamente se está asumiendo sin pérdida de generalidad que las m' primeras columnas de la matriz S son las asociadas al problema restringido.

Queremos ver que se cumplen las condiciones suficientes de optimalidad para el problema (MP). Para ello tomamos ahora el vector $\bar{y} = (\bar{x}^T, 0^T) \in \mathbb{R}^m$ y completamos el vector de multiplicadores de Lagrange: $\underline{\mu_i} = 1 - s_i^T \hat{\mu}$ para $i = m' + 1, \ldots, m$. Utilizando la hipótesis del teorema $1 - \bar{u}^T \hat{\mu} \geq 0$, se deduce la condición del signo sobre estos nuevos multiplicadores: $\underline{\mu_i} \geq 0$ para todo $i = m' + 1, \ldots, m$. Por lo tanto se ha encontrado un conjunto de multiplicadores de Lagrange asociados a $\bar{y}$ para el cual se verifican las condiciones suficientes de optimalidad, luego se puede afirmar que $\bar{y}$ es solución del (MP). □

El Teorema 3.1 establece una valiosa condición fácil de implementar computacionalmente que nos permite saber cuando una solución del (RMP) es solución del (MP), lo que ocurre cuando $c_{Red} = 1 - \bar{u}^T \hat{\mu} \geq 0$. Tal como se ha explicado, el *Pricing Problem* no es solo el que establece esa relación entre soluciones, si no que además las soluciones del (PP) son los nuevos patrones de empaquetamiento (columna de S) a añadir de forma iterativa al (RMP). Esta es la estrategia más utilizada, aunque esto no asegura una mayor velocidad de resolución. De hecho, aunque en este proyecto no se ha considerado, otra estrategia válida sería añadir cualquier patrón con coste reducido negativo al (RMP) en cada iteración.

En este punto es interesante poner lo expresado previamente en formato de pseudocódigo como se muestra en el Algoritmo 1. Si uno analiza el algoritmo, puede llegar a preguntarse si pueden darse situaciones en las que no se termina en un número finito de pasos, es decir, situaciones de bucle infinito. Esto podría producirse únicamente si el (PP) añadiera patrones ya considerados. Pues bien, la Proposición 3.2 nos asegura que estas situaciones no se producen en el Algoritmo 1.

Proposición 3.2. *Sea $\bar{x}$ la solución obtenida del (RMP) con m' variables (m' patrones), entonces el patrón dado como solución del (PP) no es ninguno de los considerados previamente en el (RMP).*

Demostración. Como $\bar{x}$ es solución del (RMP), usando el Teorema A.5 se verifica (3.6) con $\underline{\mu_i} \geq 0$ para $i = 1, \ldots, m'$. De donde los patrones que están en el (RMP) verifican:

$$1 - \sum_{j=1}^{n} \hat{\mu}_j s_{ji} \geq 0, \quad i = 1, \ldots, m'. \tag{3.7}$$

Por otro lado, los patrones que se añaden al (RMP), $\bar{u} = (s_{1k}, s_{2k}, \ldots, s_{mk})^T$, tienen coste reducido negativo: $1 - \bar{u}^T \hat{\mu} < 0$. Por tanto esos patrones no verifican (3.7) por lo que no son los que tenemos en el problema (RMP). □

Un aspecto a destacar del Algoritmo 1 es que la matriz S ha sido inicializada con un conjunto suficiente de patrones de forma que todos los rectángulos pueden ser empaquetados, por lo tanto (aplicando el Teorema A.4) la existencia de solución del (RMP) está siempre garantizada. Con todas estas consideraciones, se puede afirmar que el algoritmo está bien definido ya que hemos demostrado que no se producen bucles infinitos y siempre existen puntos admisibles.

Algoritmo 1: Generación de columnas(W, H, w, h)

Entrada: W: anchura del contenedor.
H: altura del contenedor.
w: array de las anchuras de los rectángulos.
h: array de las alturas de los rectángulos.

Salida : Lista $(\bar{x}, S, M)$ formado por la solución del (MP), $\bar{x}$, la matriz con los patrones S y la matriz de coordenadas M_C.

`/* Inicialización variables */`

S = matriz conteniendo un conjunto de patrones admisibles tal que existe solución del (RMP);

M_C = matriz que contiene las coordenadas de la esquina inferior izquierda de los rectángulos para cada patrón de S;

encontrada = False;

`/* Inicialización variables auxiliares */`

$\bar{x} = [\,]$; $\hat{\mu} = [\,]$; $\bar{u} = [\,]$; $coordenadas = [\,]$;

`/* Empiezan las iteraciones. */`

mientras *no encontrada* **hacer**
  Resolvemos el (RMP)$\rightarrow \bar{x}, \hat{\mu}$;
  Resolvemos el (PP)$\rightarrow \bar{u}, coordenadas$;
  si $1 - \bar{u}^T\hat{\mu} \geq 0$ **entonces**
    encontrada = True;
  en otro caso
    $S = [S, \bar{u}]$;
    $M_C = [M_C, coordenadas]$;
  fin
fin
devolver $(\bar{x}, S, M)$

El (RMP) es un problema de programación lineal que puede ser resuelto de forma eficiente, por ejemplo, por el programa `linprog` de Matlab, que es el que se ha empleado en este proyecto. Este se basa en la utilización del método Simplex.

Por otro lado, el (PP) formulado en (3.5) es un problema de programación entera mixta con menor número de variables y restricciones en comparación con el (2DBPP). Pese a eso, la complejidad del problema es muy alta y programas como `intlinprog` de Matlab tendrían dificultades en su resolución. Por ello, en el capítulo 4 se explica las distintas técnicas algorítmicas empleadas para resolver el (PP), así como distintas cotas inferiores empleadas a lo largo del proyecto.

Capítulo 4

Resolución del Pricing Problem y cotas inferiores

En este capítulo se presentan dos de los principales algoritmos para la resolución del Pricing Problem: Búsqueda Manteniendo Arco-Consistencia (MAC, del inglés Maintaining Arc Consistency) y Branch and Bound (Ramificación y Poda). Además, se introducen distintas cotas inferiores sobre el número de patrones necesarios para el empaquetamiento claves para este proyecto ya que son las que nos permiten concluir que la solución obtenida es óptima o no.

4.1. Descomposición del Pricing Problem en dos etapas

Como se ha expuesto en el capítulo anterior, el Pricing Problem tal como está formulado en (3.5) es muy difícil de resolver por programas de programación entera mixta como `intlinprog`. Por ello, tal como se hace en [10], en este proyecto se ha optado por dividir el Pricing Problem en dos etapas, que se repiten hasta resolver el problema.

La primera etapa corresponde a la resolución de un problema de la mochila unidimensional (1DKPP, siglas correspondientes a *one-dimensional knapsack problem* en inglés). Básicamente lo que estamos haciendo es reducir el número de variables y restricciones, de manera que inicialmente tenemos un problema de programación entera con n variables binarias y una sola restricción, que hace referencia al tamaño del contenedor:

$$(\text{1DKPP}) \begin{cases} \text{Maximizar} & \displaystyle\sum_{i=1}^{n} \hat{\mu}_i u_i \\ u \in \{0,1\}^n & \\ \text{Sujeto a:} & \displaystyle\sum_{i=1}^{n} w_i h_i u_i \leq WH. \end{cases} \tag{4.1}$$

Sea $\bar{u}$ solución del (1DKPP) y $P = \{i \in I_R : \bar{u}_i = 1\}$. En la segunda etapa resolvemos el Problema de Decisión (DP, *Decision Problem* en inglés), que consiste en comprobar si todos los rectángulos asociados al conjunto P pueden ser colocados en un único contenedor

o no. Si la respuesta es afirmativa, entonces $\bar{u}$ puede ampliarse a una solución del (PP) eligiendo los valores adecuados para las variables l_{ij}, b_{ij}, x_i e y_i. En caso contrario, $\bar{u}$ no generaría una solución del (PP) por falta de admisibilidad y habría que volver a repetir el proceso anterior añadiendo al (1DKPP) la siguiente restricción:

$$\sum_{i \in P} u_i \leq |P| - 1. \tag{4.2}$$

De esta forma, esta desigualdad que se añade al (1DKPP) impide que el patrón escogido en la iteración anterior sea escogido nuevamente. La repetición de estas dos etapas lleva eventualmente a una solución del (PP).

Sea $\mathcal{C}$ el conjunto de conjuntos de índices de rectángulos que no pueden ser colocados en un único contenedor y que han sido generados a partir de la segunda etapa. Entonces, el problema (1DKPP) se transforma en el problema de la mochila unidimensional con múltiples restricciones (1DMCKP, *one-dimensional multi-constrained knapsack problem* en inglés):

$$\text{(1DMCKP)} \begin{cases} \text{Maximizar } \displaystyle\sum_{i=1}^{n} \hat{\mu}_i u_i \\ u \in \{0,1\}^n \\ \text{Sujeto a: } \displaystyle\sum_{i=1}^{n} w_i h_i u_i \leq WH \\ \qquad\qquad \displaystyle\sum_{i \in P} u_i \leq |P| - 1, \ P \in \mathcal{C}. \end{cases} \tag{4.3}$$

Es evidente que si la solución $\bar{u}$ está asociada a un empaquetamiento de un solo contenedor, entonces ampliándola con las variables que aparecen en (3.5) tendríamos una solución del (PP).

En el Algoritmo 2 se ha descrito la forma de resolver el *Pricing Problem* en este proyecto. Las restricciones generales del (1DMCKP) incluidas en las matrices A_{KP} y b_{KP} (salvo la primera) contienen información de combinaciones de rectángulos que no pueden ser colocados en un único contenedor y estas son válidas para todos los *Pricing Problem* asociados al mismo problema, luego una vez que estas restricciones son añadidas a un (PP) se mantienen para todos los siguientes. Por consiguiente, el esquema presentado en el Algoritmo 2 ha sido programado de esta forma salvo por una ligera diferencia: la matriz A_{KP} y el vector b_{KP} han sido definidos como variables globales del Algoritmo 1. Experimentos numéricos han permitido comprobar que definiendo A_{KP} y b_{KP} como variables globales se consigue disminuir el tiempo de ejecución notablemente.

Las siguientes secciones se van a destinar a explicar cómo resolver el (1DMCKP) y el problema de decisión (DP). Para la resolución del problema de la mochila (1DMCKP) se va a emplear una técnica algorítmica conocida como Branch and Bound que se encuentra implementada a través del software de programación lineal entera mixta `intlinprog` de Matlab. En cambio, para el (DP) se va a formular como un problema de satisfacción de restricciones (CSP, *Constraint Satisfaction Problem* en inglés) y va a ser resuelto utilizando el algorimo MAC.

Algoritmo 2: Resolvedor del Pricing Problem$(W, H, w, h, \hat{\mu})$

Entrada: W: anchura del contenedor.
H: altura del contenedor.
w: array de las anchuras de los rectángulos.
h: array de las alturas de los rectángulos.
$\hat{\mu}$: vector de multiplicadores de Lagrange obtenidos del (RMP).

Salida : Lista $(\bar{u}, coordenadas)$ formado por la solución del (PP), $\bar{u}$, y las coordenadas de los rectángulos correspondientes a $\bar{u}$, *coordenadas*.

```
/* Inicialización variables                                    */
```
$A_{KP} = [w \cdot h]$; `// Matriz de restricciones del (1DMCKP).`
$b_{KP} = [W \cdot H]$; `// Vector columna de términos independientes del (1DMCKP).`
$\bar{u} = [\,]$; `// Vector columna solución del (1DMCKP).`
$coordenadas = [\,]$; `// Vector columna que contiene las coordenadas` (x, y) `de los rectángulos solución del (1DMCKP).`
solucionado = False; `// Variable que indica si se ha obtenido solución del (PP).`
```
/* Empiezan las iteraciones.                                   */
```
mientras *no solucionado* **hacer**
  Resolver el (1DMCKP)$\rightarrow \bar{u}$;
  Resolver el (DP)$\rightarrow solucionado, coordenadas$;
  si *no solucionado* **entonces**
    $A_{KP} = [A_{KP}; \bar{u}^T]$;
    $b_{KP} = [b_{KP}; suma(\bar{u}) - 1]$;
  fin
fin
devolver $(\bar{u}, coordenadas)$

4.2. Resolviendo el problema de la mochila (1DMCKP)

Consideremos el problema (1DMCKP) formulado de la siguiente forma:

$$z = \text{máx}\{\hat{\mu}^T u : u \in \mathcal{A}\}, \tag{4.4}$$

siendo $\mathcal{A}$ el conjunto de puntos admisibles, es decir, el conjunto de puntos de $\{0,1\}^n$ que verifican las restricciones del problema (4.3) [12].

Para resolver este problema se va a utilizar una técnica algorítmica conocida como Branch and Bound (ramificación y poda) [12] que consiste en dividir el problema en otros subproblemas más fáciles de resolver y aprovechar toda esa información para resolver el problema original. Esta información se puede utilizar para podar el árbol de soluciones, de forma que no sea necesario explorarlo entero y realizar una búsqueda más eficiente.

Proposición 4.1. *Sea $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_K$ una descomposición de $\mathcal{A}$ en conjuntos más pequeños y disjuntos entre sí, y sea $z^k = \text{máx}\{\hat{\mu}^T u : u \in \mathcal{A}_k\}$ para $k = 1, \ldots, K$. Entonces $z = \text{máx}\{z^k : k = 1, \ldots, K\}$.*

El proceso comienza fijando unas cotas superior e inferior de z, descomponiendo en problemas más sencillos se pretende mejorar estas a partir de las cotas de esos subproblemas. Para ello se hace uso de la siguiente propiedad:

Proposición 4.2. *Sea $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_K$ una descomposición de $\mathcal{A}$ en conjuntos más pequeños y disjuntos entre sí, y sean $\bar{z}^k$ y $\underline{z}^k$ respectivamente cotas superior e inferior de $z^k = \max\{\hat{\mu}^T u : u \in \mathcal{A}_k\}$ para $k = 1, \ldots, K$. Entonces $\bar{z} = \max\{\bar{z}^k : k = 1, \ldots, K\}$ es una cota superior de z y $\underline{z} = \max\{\underline{z}^k : k = 1, \ldots, K\}$ es una cota inferior de z.*

La información dada por las cotas superior, $\bar{z}$, e inferior, $\underline{z}$, sobre el valor óptimo z permite conocer qué conjuntos tienen que ser examinados en mayor profundidad para obtener z. Básicamente se usan para saber cuando se pueden podar ciertas ramas del árbol de soluciones.

Hay tres razones por las que se puede podar el árbol de soluciones:

1. Poda por **optimalidad**: $z^t = \max\{\hat{\mu}^T u : u \in \mathcal{A}_t\}$ ha sido resuelto, lo que ocurre cuando $\bar{z}^t = \underline{z}^t$.
2. Poda por **cota**: $\bar{z}^t \leq \underline{z}$.
3. Poda por no **admisibilidad**: $\mathcal{A}_t = \emptyset$.

Ejemplo 4.3. *Sea $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ una descomposición de $\mathcal{A}$. A continuación, se muestran tres situaciones distintas para ejemplificar la poda del árbol de soluciones, donde se pueden observar las cotas superior e inferior asociadas a los problemas correspondientes.*

SITUACIÓN 1:

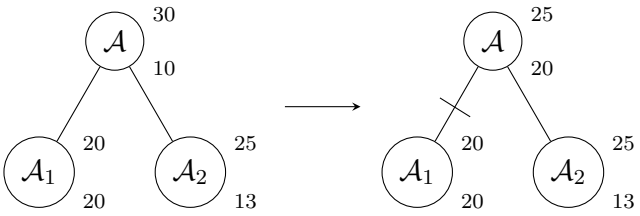


Figura 4.1: Poda por optimalidad.

Primero, en la Figura 4.1 se observa que las cotas iniciales inferior y superior que aparecen en el nodo $\mathcal{A}$ son $\bar{z} = 30$ y $\underline{z} = 10$. Utilizando ahora la Proposición 4.2 y las cotas de los subproblemas asociados a $\mathcal{A}_1$ y $\mathcal{A}_2$, se obtiene que $\bar{z} = \max_k \bar{z}^k = \max\{20, 25\} = 25$ y $\underline{z} = \max_k \underline{z}^k = \max\{20, 13\} = 20$, mejorando los valores iniciales de las cotas.

En segundo lugar, se observa que las cotas inferior y superior del nodo $\mathcal{A}_1$ coinciden, por lo que no hay razón para explorar el conjunto $\mathcal{A}_1$. Por ello, la rama asociada a $\mathcal{A}_1$ es podada por optimalidad.

SITUACIÓN 2:

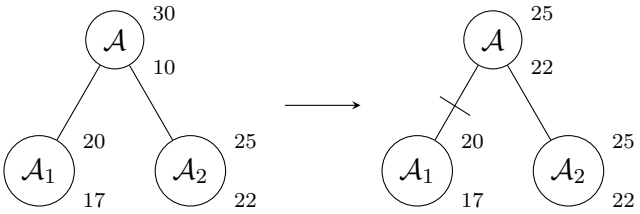


Figura 4.2: Poda por cota.

Primero, en la Figura 4.2 se observa que las cotas iniciales inferior y superior que aparecen en el nodo $\mathcal{A}$ son $\bar{z} = 30$ y $\underline{z} = 10$. Utilizando ahora la Proposición 4.2 y las cotas de los subproblemas asociados a $\mathcal{A}_1$ y $\mathcal{A}_2$, se obtiene que $\bar{z} = \text{máx}_k\, \bar{z}^k = \text{máx}\{20, 25\} = 25$ y $\underline{z} = \text{máx}_k\, \underline{z}^k = \text{máx}\{17, 22\} = 22$, mejorando los valores iniciales de las cotas.

En segundo lugar, se observa que la solución óptima tiene un valor de al menos 22 y la cota superior asociada a $\mathcal{A}_1$ es $\bar{z}^1 = 20$, por lo ninguna solución óptima puede encontrarse en el conjunto $\mathcal{A}_1$. Por ello, la rama asociada a $\mathcal{A}_1$ es podada por cota.

SITUACIÓN 3:

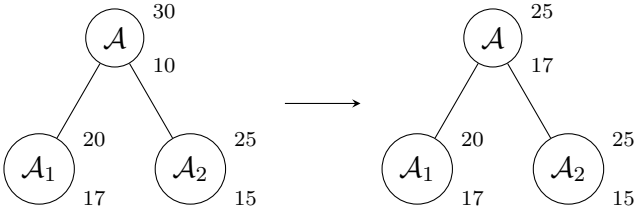


Figura 4.3: Ninguna poda es posible.

Primero, en la Figura 4.3 se observa que las cotas iniciales inferior y superior que aparecen en el nodo $\mathcal{A}$ corresponden a $\bar{z} = 30$ y $\underline{z} = 10$. Utilizando ahora la Proposición 4.2 y las cotas de los subproblemas asociados a $\mathcal{A}_1$ y $\mathcal{A}_2$, se obtiene que $\bar{z} = \text{máx}_k\, \bar{z}^k = \text{máx}\{20, 25\} = 25$ y $\underline{z} = \text{máx}_k\, \underline{z}^k = \text{máx}\{17, 15\} = 17$, mejorando los valores de las cotas. Ninguna otra conclusión puede extraerse y es necesario explorar los conjuntos $\mathcal{A}_1$ y $\mathcal{A}_2$.

En este punto ya se ha explicado las bases del algoritmo Branch and Bound, por lo que ahora veamos como se aplica esta técnica para la resolución del (1DMCKP). Se necesitan conocer tres aspectos: cómo calcular las cotas, cómo ramificar y cómo explorar los nodos.

En primer lugar, el calculo de las cotas se realiza utilizando la relajación continua LP del problema. Que en este caso consistirá en convertir las variables binarias u_i en reales y con valores entre 0 y 1. Para resolver este problema se ha usado el método Simplex que se encuentra implementado en la función `linprog` de Matlab. De esta forma, se consigue minimizar la función objetivo del (1DMCKP) pero sobre un conjunto mayor que el admisible de ese problema, por lo que la relajación continua permite obtener una cota superior de la función objetivo. Respecto a la cota inferior, esta se inicializa con un valor $\underline{z} = -\infty$ y solo se actualiza cuando se obtiene una solución entera en la relajación continua LP. El código

`intlinprog` dispone además de algoritmos heurísticos para encontrar soluciones enteras factibles, lo que permite obtener cotas inferiores más rápidamente, así como aumentar la velocidad de resolución.

Algoritmo 3: Branch and Bound($\hat{\mu}, A_{KP}, b_{KP}$)

Entrada: $\hat{\mu}$: vector de multiplicadores de Lagrange obtenidos del (RMP).
A_{KP}: matriz de restricciones del (1DMCKP).
b_{KP}: vector columna de términos independientes del (1DMCKP).
Salida : $\bar{u}$: solución del (1DMCKP).

```
/* Inicialización                                                    */
```
$Lista = [\ (1DMCKP)]$; `// Lista de problemas a resolver.`
$\underline{z} = -\infty$; `// Cota inferior del (1DMCKP).`
$\bar{u} = [\ ]$; `// Vector columna solución del (1DMCKP).`
```
/* Empiezan las iteraciones.                                         */
```
mientras *Lista sea no vacía* **hacer**
  PResolver=primer problema de Lista;
  Eliminación del primer problema de Lista;
  Resolver la relajación LP de PResolver→ se obtiene $\bar{u}_{aux}, f(\bar{u}_{aux})$;
  si *PResolver no tiene puntos admisibles* **entonces**
    Podar por no admisibilidad;
    break;
  fin
  si $f(\bar{u}_{aux}) \leq \underline{z}$ **entonces**
    Podar por cota;
    break;
  fin
  si $\bar{u}_{aux}$ *es entero* **entonces**
    $\underline{z} = f(\bar{u}_{aux})$;
    $\bar{u} = \bar{u}_{aux}$;
    Podar por optimalidad;
    break;
  fin
  Guardar $\bar{u}_{aux}$ y $f(\bar{u}_{aux})$ del problema;
  Añadir dos subproblemas a Lista generados por ramificación del nodo actual;
  Actualizar el orden de Lista, de acuerdo a la estrategia de exploración de los nodos;
fin
devolver $\bar{u}$

En segundo lugar, el árbol binario de soluciones se ramifica escogiendo una de las variables que toma un valor no entero en la solución dada por la relajación continua. Sin embargo, como puede haber varios candidatas posibles es necesario que haya una regla para escoger entre estas. Una de las reglas más comunes (y la empleada en esta sección) es escoger la variable u_j con parte fraccional más cercana a 0.5.

En caso de empate es indiferente que variable escoger. Sin embargo, experimentos numéricos realizados en [1] muestran que el rendimiento de esta regla no es mejor que seleccionar la variable aleatoriamente. Por esa razón, es más recomendable con `intlinprog`

utilizar otras reglas más sofisticadas como ramificación por pseudocoste. Una vez seleccionada la variable u_j, se produce la ramificación del conjunto $\mathcal{A}$ en los conjuntos disjuntos $\mathcal{A}_1$ y $\mathcal{A}_2$ de la siguiente forma:

$$\mathcal{A}_1 = \mathcal{A} \cap \{u \in \mathcal{A} : u_j = 0\},\ \mathcal{A}_2 = \mathcal{A} \cap \{u \in \mathcal{A} : u_j = 1\}.$$

Por último, queda hablar que el orden de exploración de los nodos no visitados, que se encuentran almacenados en una lista. De entre las distintas formas de escoger el nodo a explorar, en esta sección se va a utilizar se llama Mejor Nodo Primero y consiste en escoger el nodo con la cota superior más grande. De esta forma, nunca se ramificará un nodo cuya cota superior es menor que el valor óptimo z. En cambio, `intlinprog` utiliza por defecto una regla más sofisticada conocida como Mejor Proyección.

En `intlinprog` se realiza además un preprocesamiento del problema, lo que permite reducir su número de variables y/o restricciones, reduciendo el tiempo total de ejecución del programa.

En el Algoritmo 3 se presenta una versión simplificada del algoritmo Branch and Bound para la resolución del problema (1DMCKP). Apliquemoslo a la resolución del Ejemplo 4.4.

Ejemplo 4.4. *Resolver el siguiente problema de la mochila (P) utilizando la técnica de Branch and Bound:*

$$\text{(P)} \begin{cases} \text{Maximizar } f(u) = \hat{\mu}^T u = u_1 + u_2 + u_3 + u_4 + u_5 + u_6 \\ u \in \{0,1\}^6 \\ \text{Sujeto a:} \quad 12u_1 + 8u_2 + 9u_3 + 6u_4 + 4u_5 + 6u_6 \leq 30 \\ \qquad\qquad\quad u_3 + u_6 \leq 1 \end{cases}$$

En primer lugar hay que aplicar la relajación continua LP al problema (P). El resultado que se obtiene al resolver el problema con `linprog` de Matlab es:

$$\bar{u} = \left(\frac{1}{2}, 1, 0, 1, 1, 1\right),\ f(\bar{u}) = 4.5.$$

Por lo tanto la relajación continua sobre (P) permite obtener una cota superior $\bar{z} = 4.5$, pero no una cota inferior, ya que $\bar{u}$ no pertenece a $\{0,1\}^6$. Por esta razón, $\underline{z} = -\infty$. Ahora, como $\underline{z} < \bar{z}$ se ramifica utilizando la única variable no entera, u_1, definiendo dos subconjuntos del conjunto de puntos admisibles de (P), $\mathcal{A}$:

$$\mathcal{A}_1 = \{u \in \mathcal{A} : u_1 = 0\},\ \mathcal{A}_2 = \{u \in \mathcal{A} : u_1 = 1\}.$$

En este punto hay dos nodos por explorar. De forma arbitraria se escoge el nodo $\mathcal{A}_1$. Ahora se resuelve la relajación continua de máx$\{\hat{\mu}^T u : u \in \mathcal{A}_1\}$, lo que da lugar a la siguiente solución:

$$\bar{u}^1 = (0, 1, 1, 1, 1, 0),\ \bar{z}^1 = f(\bar{u}^1) = 4.$$

Como se ha encontrado una solución entera en la resolución del problema relajado, la exploración de este conjunto no aporta más información y se poda por optimalidad.

Ahora se explora el último nodo restante: $\mathcal{A}_2$. Se resuelve la relajación continua de máx$\{\hat{\mu}^T u : u \in \mathcal{A}_2\}$ obteniendo como solución:

$$\bar{u}^2 = \left(1, \frac{1}{4}, 0, 1, 1, 1\right), \ \bar{z}^2 = f(\bar{u}^2) = 4.25.$$

Aplicando la Proposición 4.2 y las cotas de los subproblemas asociados a $\mathcal{A}_1$ y $\mathcal{A}_2$, se obtienen las cotas $\bar{z} = \text{máx}_k\, \bar{z}^k = \text{máx}\{4, 4.25\} = 4.25$ y $\underline{z} = \text{máx}_k\, \underline{z}^k = \text{máx}\{4, -\infty\} = 4$, mejorando los valores anteriores. En la siguiente figura se muestra el estado del árbol de soluciones:

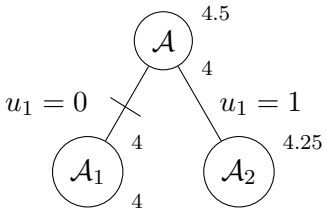


Figura 4.4: Árbol parcial generado por el algoritmo Branch and Bound tras la primera ramificación.

El siguiente paso es dividir el conjunto $\mathcal{A}_2$ utilizando la única variable que toma valor no entero en la relajación continua: u_2. Para ello, se realiza una descomposición de $\mathcal{A}_2$ en $\mathcal{A}_{21}$ y $\mathcal{A}_{22}$:

$$\mathcal{A}_{21} = \{u \in \mathcal{A} : u_1 = 1, u_2 = 0\}, \ \mathcal{A}_{22} = \{u \in \mathcal{A} : u_1 = u_2 = 1\}.$$

Hay dos nodos para explorar y ambos tienen la misma cota superior asociada, por lo que de forma arbitraria se empieza por el nodo $\mathcal{A}_{21}$. Se le aplica la relajación continua a máx$\{\hat{\mu}^T u : u \in \mathcal{A}_{21}\}$ y se resuelve usando `linprog`, obteniendo la solución:

$$\bar{u}^{21} = \left(1, 0, \frac{2}{3}, 1, 1, \frac{1}{3}\right), \ \bar{z}^{21} = f(\bar{u}^{21}) = 4.$$

En este caso como $4 = \bar{z}^{21} \leq \underline{z} = 4$, la rama asociada al nodo $\mathcal{A}_{21}$ se poda por cota. A continuación se explora el nodo restante, $\mathcal{A}_{22}$. Aplicando relajación continua LP a máx$\{\hat{\mu}^T u : u \in \mathcal{A}_{22}\}$ y resolviendo el problema utilizando `linprog` se obtiene:

$$\bar{u}^{22} = (1, 1, 0, 0, 1, 1), \ \bar{z}^{22} = f(\bar{u}^{22}) = 4.$$

Esta solución es entera, por lo $\bar{z}^{22} = \underline{z}^{22} = 4$. Como se verifica que $4 = \bar{z}^{22} \leq \underline{z} = 4$, se poda por cota la rama asociada al conjunto $\mathcal{A}_{22}$, por lo que no queda ningún nodo por explorar. Notar que se verifica que el valor que alcanza la función objetivo en $\bar{u}^{22}$ es el mismo que en $\bar{u}^1$. Ambas soluciones son igual de válidas ya que en programación entera la unicidad de solución no está garantizada. Sin embargo, como por construcción del algoritmo Branch and Bound se comprueba antes si una rama se puede podar por cota que por optimalidad, la solución que retorna el algoritmo es $\bar{u}^1$.

En la Figura 4.5 se muestra el árbol de soluciones asociado a la ejecución completa del algoritmo Branch and Bound. Sin embargo, un sencillo análisis del problema hubiera

permitido resolver el problema antes. La función objetivo bajo las restricciones de (P) toma valores enteros entre 0 y 6 exclusivamente, por esa razón, la exploración del conjunto $\mathcal{A}_2$ es innecesaria ya que la solución óptima z estaba acotada por $\bar{z} = 4.25$ y $\underline{z} = 4$, y el único valor factible es $z = 4$, del cual ya se había obtenido un punto admisible de (P).

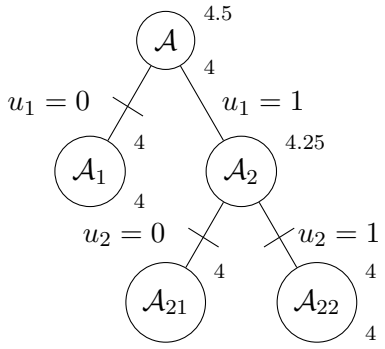


Figura 4.5: Árbol completo generado por el algoritmo Branch and Bound.

4.3. Resolviendo el problema de decisión (DP)

En esta sección se va a explicar cómo se resuelve el problema de decisión (DP) que comprueba si un conjunto de rectángulos dado puede ser colocado en un único contenedor o no. Para ello, se va a formalizar como un problema de satisfacción de restricciones (CSP, *Constraint Satisfaction Problem* en inglés) lo que permite representar el problema de forma simple y estructurada.

Definición 4.5. *Un **problema de satisfacción de restricciones (CSP)** está definido por tres componentes:*

- *Un conjunto de **variables** $\{r_1, r_2, \ldots, r_n\}$.*
- *Un conjunto de **dominios** $\{D_1, D_2, \ldots, D_n\}$. Cada variable r_i tiene asociado un conjunto finito no vacío D_i (llamado dominio) que contiene los posibles valores que puede tomar esta.*
- *Un conjunto de **restricciones** $\{C_1, C_2, \ldots, C_m\}$ sobre los valores de las variables.*

Definición 4.6. *Un **estado** de un CSP es una **asignación** de valores a algunas o todas las variables. Una asignación puede ser, según las variables consideradas:*

- ***Parcial**: asigna valores a algunas de las variables del problema.*
- ***Total**: asigna valores a todas las variables del problema.*

También será, según las restricciones que cumpla:

- ***Factible** o **consistente**: si el estado cumple todas las restricciones del CSP.*

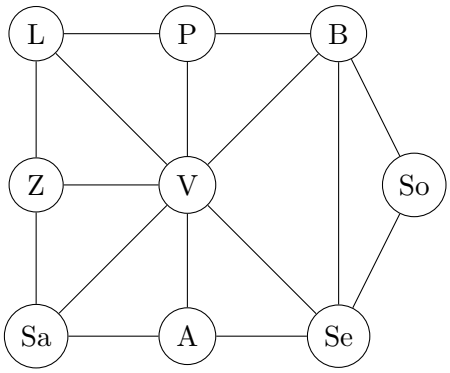
- ***No factible*** *o* ***inconsistente****: si el estado viola alguna restricción del CSP.*

Definición 4.7. *Una* ***solución*** *del CSP es una asignación total y factible.*

Como es extremadamente complicado presentar el modelo CSP para nuestro problema de empaquetamiento, ahora vamos a considerar un ejemplo en otro ámbito más sencillo de formular como es el coloreado de mapas y que se muestra a continuación.

Ejemplo 4.8. *Colorear el mapa de provincias de Castilla y León utilizando cuatro colores (naranja, verde, azul y morado) de forma que dos provincias limítrofes no puedan estar coloreadas del mismo color.*

Lo primero que hacemos es convertir el mapa de provincias de Castilla y León en un grafo al que llamaremos $G = (V, E)$, donde V es el conjunto de nodos del grafo que se corresponden a las variables del problema y E es el conjunto de arcos que corresponden a las restricciones del CSP. A partir del grafo G podemos modelar el problema como un CSP determinando las variables, dominios y restricciones involucradas:



- **Variables**:

  $\{L, P, B, Z, V, So, Sa, A, Se\}$.

- **Dominios**:

  $D_L = \cdots = D_{Se} = \{n, v, a, m\}$.

- **Restricciones**:

  $\{r_i \neq r_j : (r_i, r_j) \in E\}$.

Figura 4.6: Grafo del mapa de provincias de Castilla y León y el modelado del problema como un CSP.

Una posible solución al problema es la siguiente asignación:

$$L = n, P = v, B = a, Z = v, V = m, So = v, Sa = n, A = v, Se = n.$$

A continuación, volviendo a nuestro problema de empaquetamiento se muestra como formular el problema de decisión (DP) como un CSP. Para ello asociamos a cada par de rectángulos $i, j \in I_R$ con $i < j$ la variable de relación r_{ij} perteneciente al dominio $D_{ij} = \{izquierda, derecha, abajo, arriba\}$, que establece las posición relativa entre los rectángulos i y j, indicando la posición del primero respecto del segundo. Para evitar obtener soluciones simétricas, los rectángulos 1 y 2 tienen asociado el dominio restringido $D_{12} = \{izquierda, abajo\}$. Por último, las restricciones asociadas a las distintas variables de relación establecen las condiciones de no solapamiento y las asociadas a las dimensiones

del contenedor.

$$
\text{(CSP)} \begin{cases}
\bullet \textbf{ Variables:} \\
\quad \{r_{ij} : i, j \in I_R, i < j\}. \\
\bullet \textbf{ Dominios:} \\
\quad D_{12} = \{izquierda, abajo\}, \\
\quad D_{ij} = \{izquierda, derecha, abajo, arriba\},\ i, j \in I_R,\ i < j. \\
\bullet \textbf{ Restricciones:} \\
\quad r_{ij} = izquierda \Rightarrow x_i + w_i \leq x_j,\ i, j \in I_R, i < j, \\
\quad r_{ij} = derecha \Rightarrow x_j + w_j \leq x_i,\ i, j \in I_R,\ i < j, \\
\quad r_{ij} = abajo \Rightarrow y_i + h_i \leq y_j,\ i, j \in I_R,\ i < j, \\
\quad r_{ij} = arriba \Rightarrow y_j + h_j \leq y_i,\ i, j \in I_R,\ i < j, \\
\quad 0 \leq x_i \leq W - w_i,\ i \in I_R, \\
\quad 0 \leq y_i \leq H - h_i,\ i \in I_R.
\end{cases} \tag{4.5}
$$

Nótese que el número de variables de relación r_{ij} es $\frac{n^2-n}{2}$ ya que se verifica que si $r_{ij} = \eta$, entonces $r_{ji} = \neg\eta$ para cualquier i y j de I_R y η de D_{ij}. Al igual que ocurría en (2.15), nosotros estamos interesados en particular en las coordenadas (x, y) de cada rectángulo, y no tanto en las variables de r_{ij}. De hecho, las restricciones afectan a las variables x e y de las coordenadas de los rectángulos.

El problema formulado en (4.5) se resuelve utilizando el algoritmo conocido como Búsqueda Manteniendo Arco-Consistencia (MAC, *Mantaining Arc Consistency* en inglés). Este algoritmo es la combinación de la búsqueda en profundidad con vuelta atrás (búsqueda con bactracking) y la arco-consistencia, lo que permite podar dominios.

La búsqueda en profundidad con vuelta atrás está representada por un árbol de búsqueda donde cada nodo corresponde a una solución parcial en la que son fijados los valores de algunas variables. La búsqueda en profundidad se corresponde con una estructura de pila (LIFO), donde el último nodo añadido a la frontera (nodos que esperan a ser visitados) es el primero que se expande. El término búsqueda con vuelta atrás se utiliza para la búsqueda en profundidad que elige valores para una variable a la vez y vuelve atrás cuando una variable no tiene ningún valor legal para asignarle. Esto ocurre cuando algún dominio queda vacío al aplicar arco consistencia.

Todas las soluciones (en el caso de existir) se encuentran a profundidad $\frac{n^2-n}{2}$, correspondiente a tener todas las variables r_{ij} asignadas. Además, los CSP poseen una propiedad crucial: la conmutatividad.

Definición 4.9. *Un problema es* ***conmutativo*** *si el orden de aplicación de cualquier conjunto de acciones no tiene ningún efecto sobre el resultado.*

Esto ocurre en los CSP porque asignando valores a variables, alcanzamos la misma asignación parcial sin tener en cuenta el orden. Por consiguiente, todos los algoritmos de búsqueda en un CSP generan los sucesores considerando asignaciones posibles para solo una variable en cada nodo del árbol de búsqueda. Teniendo esto en cuenta, el número de hojas (nodos sin hijos) del árbol de búsqueda es de $4^m/2$, siendo $m = \frac{n^2-n}{2}$ [11].

Definición 4.10. *Un arco del (CSP) es* ***arco-consistente*** *si para cada valor de uno de sus nodos existe al menos un valor del otro nodo tal que se cumple la restricción asociada al arco. En el caso de que todos los arcos del CSP cumplan esa propiedad decimos que el CSP es* ***arco-consistente****.*

La arco-consistencia permite reducir el tamaño de los dominios, de forma que aquellas relaciones para las que no puedan encontrarse una colocación factible de rectángulos son eliminadas del dominio de D_{ij}. En caso de que $|D_{ij}| = 1$, entonces a r_{ij} se le asigna el único valor posible del dominio. Por último, en el caso de que un dominio se quede vacío, el problema no puede satisfacerse, luego volvemos hacia atrás (*backtracking*).

Aunque el algoritmo MAC hace que la complejidad temporal de cada nodo del árbol de búsqueda crezca considerablemente, el número de nodos investigados disminuye correspondientemente. En particular, la utilización del algoritmo MAC es rentable cuando se tratan problemas difíciles, como los tratados en este proyecto.

En este proyecto se utiliza el algoritmo MAC programado en C en la función `recpack` del código desarrollado en [7] con el fin de resolver un problema de empaquetamiento tridimensional, utilizando una técnica distinta a la empleada en [10]. En este proyecto ha sido necesario modificar el código disponible para ejecutar exclusivamente el algoritmo MAC para un conjunto de rectángulos y un contenedor dados, y dar solución al problema de decisión (DP). Además, en caso de que se encuentre una disposión de los rectángulos en un solo contenedor, el programa devuelve las coordenadas de los rectángulos.

Entre las modificaciones introducidas en el código original, se han creado dos nuevas funciones `CSP_solver` y `mexFunction`. La primera función, `CSP_solver`, es una adaptación de la función `binpack3d`, de forma que ejecute solamente el algoritmo MAC. La segunda función, `mexFunction`, es la interfaz del código C con Matlab, lo que permite ejecutarlo desde Matlab e integrarlo con el resto del programa. Otra modificación importante está relacionado con el hecho de que queremos resolver un problema de decisión bidimensional utilizando un algoritmo diseñado para resolver un problema tridimensional. Para adaptar el algoritmo se ha fijado la altura del contenedor a la unidad, así como la altura de todos los objetos.

El algoritmo `CSP_solver` está programado como sigue. En primer lugar se introducen los datos correspondientes a los tamaños de los rectángulos y del contenedor y se ordenan los rectángulos de mayor a menor área. A continuación, se inicializan todas las relaciones (a indefinido) y dominios. En este momento, se empiezan a realizar las llamadas recursivas al algoritmo `recpack`, en el cual las variables de relación r_{ij} son asignadas en el árbol de búsqueda de acuerdo a la siguiente ordenación, $\preceq$:

$$\forall (i,j),(i',j') \in I_R^2,\ (i,j) \preceq (i',j') \Leftrightarrow j < j' \ \vee \ (j = j' \ \wedge \ i \leq i'). \tag{4.6}$$

El algoritmo recursivo `recpack` funciona de la siguiente manera. El método recibe un par de rectángulos i y j y una relación del dominio D_{ij} como parámetros. En primer lugar, a la variable r_{ij} se le asigna la relación pasada como parámetro y se comprueba, usando la función `findcoordinates`, si existe una asignación de coordenadas de los rectángulos tal que se verifiquen las restricciones actuales. Esta función tiene un coste temporal $O(n^2)$, siendo n el número de rectángulos considerados. Dada una relación r_{ij}, `findcoordinates` modifica las coordenadas del rectángulo i o j de forma que se verifique la relación dada.

Por último comprueba que el rectángulo que se ha movido no se sale del contenedor, lo que implicaría que dicha asignación no es posible y se produciría backtracking del algoritmo MAC. En caso de encontrar dicha asignación, `recpack` procedería a la reducción de dominios por medio de la aplicación de la arco consistencia, aplicada por la función `reducedomain`. Si al reducir dominios alguno de ellos se queda vacío, entonces se produce backtracking. En caso contrario, el algoritmo `recpack` se llama de forma recursiva. La ejecución del algoritmo `CSP_solver` finaliza con una respuesta positiva cuando ha sido posible asignar un valor a todas las relaciones y el algoritmo `findcoordinates` ha podido encontrar una asignación admisible de los rectángulos en un solo contenedor. En ese caso, el algoritmo retorna un indicador con valor 1 y las coordenadas de las esquinas inferiores izquierda de los rectángulos en el contenedor. En caso de que el algoritmo recursivo no encuentre un empaquetamiento admisible, el algoritmo `CSP_solver` retorna un indicador con valor 0.

Todas las funciones implementadas en [7], han sido resumidas y simplificadas en los pseudocódigos 4 y 5. Muchas de las estructuras de datos implementadas en el código, así como algunas de las funciones auxiliares desarrolladas han sido omitidas para facilitar el entendimiento del algoritmo MAC.

Algoritmo 4: CSP_solver(h, w, H, W)

Entrada: h: array de las alturas de los rectángulos.
w: array de las anchuras de los rectángulos.
H: altura del contenedor.
W: anchura del contenedor.

Salida : Lista $(ind, coordenadas)$ formado por un indicador, ind, de si existe o no solución del (DP) y las coordenadas de los rectángulos correspondientes a la solución del (DP) obtenida, $coordenadas$.

`/* Inicialización */`

Rellenado de las estructuras (consideradas como variables globales) que almacenan la información del problema. En particular se almacena la lista $(ind, coordenadas)$.

Ordenación de los rectángulos de mayor a menor área.

$ind = 0$;

$coordenadas = [\ ]$;

$r_{ij} =$ `UNDEF`;

$D_{12} = \{izquierda, abajo\}$; $D_{ij} = \{izquierda, derecha, abajo, arriba\}$;

`/* Llamada al algoritmo recursivo recpack */`

recpack(1, 1, $length(w)$, `UNDEF`);

devolver $(ind, coordenadas)$

Algoritmo 5: recpack(i, j, n, rel)

Entrada: i, j: índices de dos rectángulos del problema.
n: número de rectángulos del problema.
rel: relación que se impone entre los rectángulos i y j.

`/* Asignación de la relación. */`
$r_{ij} = rel$;
`/* Comprobación de la existencia de una asignación de coordenadas de los rectángulos tal que se verifican todas las restricciones del problema. Si tal asignación no existe, entonces se hace backtracking. */`
Ejecución del algoritmo `findcoordinates`$\rightarrow feas$;
si *no feas* **entonces**
  return;
fin
`/* Si la relación asignada es la última, entonces se ha encontrado una solución al problema. */`
si *(i == n-1) y (j == n)* **entonces**
  Solución encontrada$\rightarrow ind = 1$.
  return;
fin
`/* Si no se ha encontrado aún una solución, se aplica arco consistencia para podar dominios. */`
Ejecución del algoritmo `reducedomain`$\rightarrow feas$;
`/* Si ningún dominio se queda vacío, entonces se llama recursivamente al algoritmo recpack. */`
si *feas* **entonces**
  $i = i + 1$;
  si $i \geq j$ **entonces**
    $i = 1$; $j = j + 1$;
  fin
  si $izquierda \in D_{ij}$ **entonces**
    recpack(i, j, n, $izquierda$);
  fin
  si $derecha \in D_{ij}$ **entonces**
    recpack(i, j, n, $derecha$);
  fin
  si $abajo \in D_{ij}$ **entonces**
    recpack(i, j, n, $abajo$);
  fin
  si $arriba \in D_{ij}$ **entonces**
    recpack(i, j, n, $arriba$);
  fin
fin
Recuperación del estado inicial con el que se ejecutó el algoritmo recpack.

A continuación, con la finalidad de ilustrar el funcionamiento del algoritmo MAC, se procede a resolver el siguiente ejemplo.

Ejemplo 4.11. *Sea el conjunto de rectángulos* $I_R = \{1, 2, 3, 4\}$ *tal que las anchuras y alturas correspondientes vienen dadas por los siguientes vectores* $w = (2, 4, 2, 2)$ *y* $h = (3, 1, 2, 1)$*. El objetivo es resolver el (DP) para un contenedor cuadrado de dimensiones* $W = H = 4$*, y en caso de que sea posible colocar todos los objetos en un solo contenedor, dar las coordenadas de los rectángulos.*

En primer lugar ordenamos los rectángulos de mayor a menor área. En este ejemplo ya están ordenados, por lo que esta etapa puede saltarse. En caso de empate nos da igual cual de los empatados sea colocado antes.

A continuación se procede la inicialización de las seis variables y seis dominios de la formulación CSP. Las variables vienen ordenadas por el siguiente vector $(r_{12}, r_{13}, r_{23}, r_{14}, r_{24}, r_{34})$ y se inicializan con valor indefinido (`UNDEF` en el pseudocódigo anterior). Los dominios en su configuración inicial son:

$$D_{12} = \{izquierda, abajo\},$$
$$D_{13} = D_{23} = D_{14} = D_{24} = D_{34} = \{izquierda, derecha, abajo, arriba\}.$$

En este punto se realiza la **primera llamada** al algoritmo recursivo `recpack`. Esta primera llamada constituye la raíz del árbol de búsqueda y por ello no se fija un valor a ninguna variable. Lo que sí se ejecuta es el algoritmo `reducedomain` para aplicar arco consistencia y podar dominios. Para representar los dominios y los valores pertenecientes a cada uno de ellos, se va a optar por utilizar un formato tabla donde aparecerá representado con una T de *True* en el caso de que el valor pertenezca al dominio dado y con una F de *False* en el caso de que dicho valor haya sido eliminado del dominio al aplicar arco consistencia. Cuando una relación haya sido fijada se redondeará con un círculo. El resultado tras aplicar arco consistencia se muestra en la siguiente tabla:

	Izquierda	Derecha	Abajo	Arriba
D_{12}	F	–	T	–
D_{13}	T	T	F	F
D_{23}	F	F	F	T
D_{14}	T	T	F	F
D_{24}	F	F	F	T
D_{34}	F	F	T	T

Tabla 4.1: Dominios tras la aplicación de la arco consistencia en la primera iteración de `recpack`.

Para la aplicación de la arco consistencia, se fija una de las relaciones sin definir (en esta iteración todas están sin definir), y se comprueba con el algoritmo `findcoordinates` si existe una asignación de coordenadas de los rectángulos tal que se verifican todas las restricciones. Para ejemplificar esto, se va a mostrar que ocurriría con los casos $r_{12} = izquierda$ y $r_{12} = abajo$. En ambas situaciones se empieza con todas las coordenadas x e y de los rectángulos inicializadas a 0.

Para el caso $r_{12} = izquierda$, se calcula una variable $sum = x_1 + w_1 = 0 + 2 = 2$. Como se verifica que $0 = x_2 < sum = 2$, entonces se asigna el valor de la variable sum a x_2 por lo que $x_2 = 2$. Ahora que ya está el rectángulo 1 a la izquierda del 2, se comprueba si el rectángulo 2 se sale del contenedor. Como $6 = 2 + 4 = x_2 + w_2 > W = 4$, entonces el rectángulo 2 se sale del contenedor y, por tanto, no es una colocación admisible de rectángulos. Consecuentemente, se elimina $izquierda$ del dominio D_{12}.

Para el caso $r_{12} = abajo$, se calcula una variable $sum = y_1 + h_1 = 0 + 3 = 3$. Como se verifica que $0 = y_2 < sum = 3$, entonces se asigna el valor de la variable sum a y_2 por lo que $y_2 = 3$. Ahora que ya está el rectángulo 1 abajo del 2, se comprueba si el rectángulo 2 se sale del contenedor. Como $4 = 3 + 1 = y_2 + h_2 = H = 4$, entonces el rectángulo 2 se no se sale del contenedor y, por tanto, es una colocación admisible de rectángulos. Consecuentemente, el valor $abajo$ permanece en el dominio D_{12}.

Esto, que se ha hecho tan solo para dos casos, se haría para todos los variables de los dominios hasta dejar el problema arco consistente. Notese que la aplicación de la arco consistencia en la primera iteración deja dominios con un solo elemento, como es el caso de D_{12} y D_{23}. En el algoritmo `reducedomain`, se asignan las relaciones correspondientes a las únicos valores disponibles de los dominios por la propia construcción de `reducedomain`. Sin embargo, cuando realmente cuenta la asignación de un valor a la variable de relación es cuando ocurre en la ejecución del algoritmo `recpack`.

Una vez que se tiene el problema arco consistente, se llama otra vez al algoritmo `recpack`. Al hacer esto se fija la relación $r_{12} = izquierda$. Tras aplicar arco consistencia no se poda ningún dominio ya que este es el único valor posible para ser escogido en el dominio D_{12}.

A continuación, se vuelve a llamar recursivamente a `recpack`, fijando esta vez la relación $r_{13} = izquierda$ y se podan el resto de dominios. El resultado de estas dos últimas iteraciones se puede observar en la tabla siguiente:

	Izquierda	Derecha	Abajo	Arriba
D_{12}	F	–	(T)	–
D_{13}	(T)	T	F	F
D_{23}	F	F	F	T
D_{14}	T	F	F	F
D_{24}	F	F	F	T
D_{34}	F	F	T	T

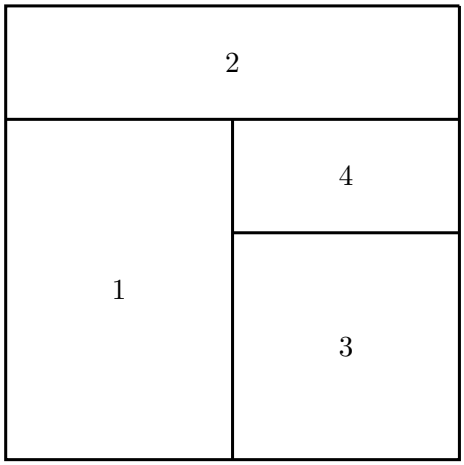
Tabla 4.2: Dominios tras la aplicación de la arco consistencia tras las segunda y tercera iteración de `recpack`. Con un círculo se muestran las relaciones fijadas.

Las siguientes tres llamadas recursivas al algoritmo recursivo `recpack` simplemente sirven para fijar las relaciones, ya que los dominios involucrados tienen un solo valor. En la última llamada recursiva se fija $r_{34} = abajo$ y se obtiene una colocación admisible de los rectángulos en un solo contenedor al ejecutar el algoritmo `findcoordinates`. Todas las relaciones tienen un valor asignado y se verifican todas las restricciones, luego se ha encontrado una solución del problema (DP). El resultado se muestra en la Tabla 4.3 y las coordenadas de los distintos rectángulos junto con un dibujo del patrón que se forma en

el contenedor se puede observar en la Figura 4.7.

	Izquierda	Derecha	Abajo	Arriba
D_{12}	F	–	(T)	–
D_{13}	(T)	T	F	F
D_{23}	F	F	F	(T)
D_{14}	(T)	F	F	F
D_{24}	F	F	F	(T)
D_{34}	F	F	(T)	T

Tabla 4.3: Dominios tras la aplicación de la arco consistencia tras la séptima iteración de `recpack`. Con un círculo se muestran las relaciones fijadas.



Coordenadas de los rectángulos (esquina inferior izquierda)

$$\begin{aligned} R_1 &\rightarrow (0,0) \\ R_2 &\rightarrow (0,3) \\ R_3 &\rightarrow (2,0) \\ R_4 &\rightarrow (2,2) \end{aligned}$$

Figura 4.7: Solución obtenida por el algoritmo MAC en la resolución del (DP).

4.4. Cotas inferiores del (2DBPP)

La cota inferior más sencilla sobre el número de contenedores necesarios para el empaquetamiento es la conocida como cota inferior continua, L_0, y es obtenida teniendo en cuenta las áreas de los rectángulos a empaquetar y el área de cada contenedor:

$$L_0 = \left\lceil \frac{\sum_{j=1}^{n} h_j w_j}{HW} \right\rceil, \tag{4.7}$$

y puede ser calculada en tiempo lineal $O(n)$.

Cotas más sofisticadas que la L_0 pueden ser obtenidas utilizando cotas derivadas de las del problema unidimensional de empaquetamiento tal como se muestra en [8] y como se explica a continuación.

Consideremos el conjunto de rectángulos $I^W = \{i \in I_R : w_i > W/2\}$. Es evidente que dos rectángulos de I^W no pueden ser empaquetados uno al lado del otro en un contenedor.

Dado cualquier entero p verificando que $1 \leq p \leq H/2$ y sean los conjuntos de rectángulos siguientes

$$I_1 = \{i \in I^W : h_i > H - p\}, \tag{4.8}$$

$$I_2 = \{i \in I^W : H - p \geq h_i > H/2\}, \tag{4.9}$$

puede observarse que dos rectángulos cualesquiera de $I_1 \cup I_2$ no pueden ser empaquetados en el mismo contenedor, por lo que $|I_1 \cup I_2|$ es una cota inferior válida (independiente del valor de p) del valor óptimo de la función objetivo del (2DBPP). Ahora sea

$$I_3 = \{i \in I^W : H/2 \geq h_i \geq p\}. \tag{4.10}$$

La cota inferior puede ser mejorada si observamos que ningún rectángulo de I_3 puede ser colocado en un contenedor usado por una pieza de I_1. Entonces, cada rectángulo de I_3 va o bien en un contenedor que tiene ya un rectángulo de I_2 o bien en otro tipo de contenedor. Como resultado de esto se pueden deducir los siguientes enunciados:

Teorema 4.12. *Dado cualquier entero p verificando que $1 \leq p \leq H/2$, una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_1^W(p) = \text{máx}\{L_\alpha^W(p), L_\beta^W(p)\}, \tag{4.11}$$

donde

$$L_\alpha^W(p) = |I_1 \cup I_2| + \text{máx}\left\{0, \left\lceil \frac{\sum_{i \in I_3} h_i - (|I_2|H - \sum_{i \in I_2} h_i)}{H} \right\rceil \right\}, \tag{4.12}$$

$$L_\beta^W(p) = |I_1 \cup I_2| + \text{máx}\left\{0, \left\lceil \frac{|I_3| - \sum_{i \in I_2} \left\lfloor \frac{H - h_i}{p} \right\rfloor}{\left\lfloor \frac{H}{p} \right\rfloor} \right\rceil \right\}. \tag{4.13}$$

Corolario 4.13. *Una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_1^W = \max_{1 \leq p \leq H/2} \{L_1^W(p)\}, \tag{4.14}$$

y puede ser calculada en tiempo $O(n^2)$.

Ahora sea $I^H = \{i \in I_R : h_i > H/2\}$. Es evidente que los resultados previos producen una cota inferior análoga

$$L_1^H = \max_{1 \leq p \leq W/2} \{L_\alpha^H(p), L_\beta^H(p)\}, \tag{4.15}$$

donde $L_\alpha^H(p)$ y $L_\beta^H(p)$ son obtenidas a través de (4.8)-(4.14) sustituyendo I^W por I^H, h_i por w_i y H por W. Teniendo todo esto en cuenta una cota inferior puede ser calculada en tiempo $O(n^2)$:

$$L_1 = \text{máx}\{L_1^W, L_1^H\}. \tag{4.16}$$

Utilizando la cota inferior L_1 se puede determinar una nueva cota L_2 que domina a las cotas L_0 y L_1. Ahora, dado un entero q verificando que $1 \leq q \leq W/2$ y sean los conjuntos de rectángulos siguientes

$$K_1 = \{i \in I_R : w_i > W - q\}, \tag{4.17}$$

$$K_2 = \{i \in I_R : W - q \geq w_i > W/2\}, \tag{4.18}$$

$$K_3 = \{i \in I_R : W/2 \geq w_i \geq q\}, \tag{4.19}$$

puede observarse que $K_1 \cup K_2 = I^W$ (independiente de q), que L_1^W es una cota inferior para I^W y que ningún rectángulo de K_3 puede colocarse junto a los de K_1. Por lo tanto, sumando a L_1^W una cota inferior sobre los contenedores necesarios para empaquetar los contenedores de K_3 se obtienen los siguientes resultados:

Teorema 4.14. *Dado un entero q verificando que $1 \leq q \leq W/2$, una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_2^W(q) = L_1^W + \text{máx}\left\{0, \left\lceil \frac{\sum_{i \in K_2 \cup K_3} h_i w_i - (HL_1^W - \sum_{i \in K_1} h_i)W}{HW} \right\rceil \right\}. \tag{4.20}$$

Corolario 4.15. *Una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_2^W = \max_{1 \leq q \leq W/2} \{L_2^W(q)\}, \tag{4.21}$$

y puede ser calculada en tiempo $O(n^2)$.

Nótese aunque el cálculo de L_1^W y de L_2^W tiene un coste temporal cuadrático $O(n^2)$ en ambos casos, el cálculo de la cota L_2^W supone un coste temporal del doble que el de la cota L_1^W, ya que para computar la primera se requiere haber calculado previamente la segunda. Teniendo en cuenta estos resultados se puede determinar de la siguiente forma la cota L_2:

Corolario 4.16. *Una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_2 = \text{máx}\{L_2^W, L_2^H\}, \tag{4.22}$$

donde L_2^H es obtenida a partir de (4.17)-(4.21) *remplazando W por H, w_i por h_i y L_1^W por L_1^H.*

Finalmente se presenta una cota inferior que es computacionalmente más cara, pero que en algunas situaciones puede mejorar los resultados de las cotas anteriores. Sean cualquier par de enteros (p, q) verificando que $1 \leq p \leq H/2$ y $1 \leq q \leq W/2$ se definen:

$$N_1 = \{i \in I_R : h_i > H - p \text{ y } w_i > W - q\}, \tag{4.23}$$

$$N_2 = \{i \in I_R \backslash N_1 : h_i > H/2 \text{ y } w_i > W/2\}, \tag{4.24}$$

$$N_3 = \{i \in I_R : H/2 \geq h_i \geq p \text{ y } W/2 \geq w_i \geq q\}. \tag{4.25}$$

Se puede apreciar que $N_1 \cup N_2$ es independiente de (p, q), que dos rectángulos cualesquiera de $N_1 \cup N_2$ no pueden ser empaquetados en el mismo contenedor y que ninguna pieza de N_3 puede colocarse en un contenedor que contenga una pieza de N_1. Una cota inferior válida puede ser determinada añadiendo a $|N_1 \cup N_2|$ el mínimo número de contenedores necesarios para aquellos rectángulos de N_3 que no pueden ser empaquetados en los contenedores usados por rectángulos de N_2. Se va a determinar una cota inferior del último valor considerando que cada rectángulo i de N_3 tiene el tamaño mínimo, es decir, $h_i = p$ y $w_i = q$. Para ello se presentan los siguientes resultados:

Lema 4.17. *Sea un contenedor de dimensiones $H \times W$ conteniendo un rectángulo de dimensiones $h_i \times w_i$, el máximo número de rectángulos de dimensiones $p \times q$ que pueden ser empaquetados en un contenedor es*

$$m(i,p,q) = \left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W - w_i}{q} \right\rfloor + \left\lfloor \frac{W}{q} \right\rfloor \left\lfloor \frac{H - h_i}{p} \right\rfloor - \left\lfloor \frac{H - h_i}{p} \right\rfloor \left\lfloor \frac{W - w_i}{q} \right\rfloor. \tag{4.26}$$

Teorema 4.18. *Dados un par de enteros (p,q) verificando que $1 \leq p \leq H/2$ y $1 \leq q \leq W/2$, una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_3(p,q) = |N_1 \cup N_2| + \text{máx}\left\{0, \left\lceil \frac{|N_3| - \sum_{i \in N_2} m(i,p,q)}{\left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W}{q} \right\rfloor} \right\rceil \right\}. \tag{4.27}$$

La cota inferior L_3 viene dada por el siguiente resultado:

Corolario 4.19. *Una cota inferior válida sobre el valor óptimo de la función objetivo del (2DBPP) es*

$$L_3 = \max_{1 \leq p \leq H/2, 1 \leq q \leq W/2} \{L_3(p,q)\}, \tag{4.28}$$

y puede ser calculada en tiempo $O(n^3)$.

Para concluir, se define una cota inferior L_4 que domina a las cuatro cotas explicadas previamente en esta sección y que viene dada por la siguiente ecuación:

$$L_4 = \text{máx}\{L_2, L_3\}. \tag{4.29}$$

Capítulo 5

Resultados numéricos

5.1. Resolviendo el modelo general

Esta sección supone una primera aproximación a la resolución numérica del problema (2.15). El objetivo de esta primera parte es intentar resolver un conjunto de problemas utilizando un software genérico de resolución de problemas de programación entera mixta. Esta manera de abordar el problema está motivada por dos razones. La primera de ellas es comprobar la afirmación que se realiza en [10], en donde se dice que el modelo es muy difícil de resolver con este tipo de algoritmos, mientras que la segunda razón es la de motivar la búsqueda de un método numérico de resolución alternativo que sea más eficiente y rápido. Para esta parte se ha optado por utilizar el programa `intlinprog` de Matlab, que implementa un método numérico general de resolución de problemas de programación lineal entera mixta [4]. Además, se han seleccionado un conjunto de doce problemas conocidos por el nombre NGCUT. Estos problemas son de generación aleatoria y fueron originalmente propuestos en [2]. Para este trabajo han sido descargados directamente del repositorio disponible en la pagina web del Grupo de Investigación Operativa de la Universidad de Bolonia [9].

Las razones para escoger este conjunto de problemas son la talla pequeña (ya que el número máximo de rectángulos involucrados es 22 como se puede observar en la Tabla 5.1) y que han sido resueltos de forma óptima en [8] (por lo que se dispone de información sobre el resultado y el tiempo de ejecución del algoritmo del artículo en cuestión). En la Tabla 5.1 se recogen de la primera a la última columna: el identificador del problema, el número de rectángulos involucrados, el tiempo de ejecución del algoritmo del artículo [8], el valor del mínimo número de contenedores obtenidos en el artículo [8], y el valor de las cotas inferiores L_0 y L_4 para los distintos problemas.

En la Tabla 5.2 se muestran los resultados numéricos obtenidos al resolver los problemas NGCUT con la formulación matricial de la sección 2.2 y el código `intlinprog`, obtenidos con la limitación computacional de un tiempo máximo de 2000 s para cada problema. Se presentan el tiempo de ejecución, el número de contenedores necesarios y la cota inferior determinada por `intlinprog` de Matlab, C_M. En rojo se han señalado los casos en los que se ha llegado al máximo tiempo de ejecución fijado y aquellos en los que tras finalizar dicho tiempo, se obtuvo un resultado de contenedores mayor que el de [8]. Tras analizar

la Tabla 5.2 se puede comprobar que solamente en tres de los doce problemas estudiados terminó, detectando una solución óptima. Además, de los otros nueve casos, en cuatro problemas no obtuvo la solución óptima, mientras que en los cinco restantes se detuvo estando en una solución óptima, pero sin haber explorado todo el árbol de búsqueda para tener la certeza de que no existían soluciones mejores.

Problema	n	Tiempo articulo / s	N_C artículo	L_0	L_4
NGCUT1	10	0,13	3	2	2
NGCUT2	17	0,85	4	3	3
NGCUT3	21	2,38	3	3	3
NGCUT4	7	0,01	2	2	2
NGCUT5	14	0,01	3	3	3
NGCUT6	15	68,98	3	2	2
NGCUT7	8	0,01	1	1	1
NGCUT8	13	0,01	2	2	2
NGCUT9	18	0,67	3	3	3
NGCUT10	13	0,01	3	2	3
NGCUT11	15	3,53	2	2	2
NGCUT12	22	2,07	3	3	3

Tabla 5.1: Resultados de los doce problemas NGCUT obtenidos por [8].

Problema	n	Tiempo intlinprog / s	N_C intlinprog	C_M
NGCUT1	10	2000,00	3	1,3
NGCUT2	17	2000,00	4	2
NGCUT3	21	2000,00	4	2
NGCUT4	7	1,18	2	1
NGCUT5	14	2000,00	3	1
NGCUT6	15	2000,00	3	1
NGCUT7	8	0,06	1	1
NGCUT8	13	32,92	2	1
NGCUT9	18	2000,00	5	1
NGCUT10	13	2000,00	3	1,6
NGCUT11	15	2000,00	3	1
NGCUT12	22	2000,00	6	1

Tabla 5.2: Resultados de la ejecución de `intlinprog` para las 12 instancias NGCUT (sin punto inicial).

Es evidente que el algoritmo del artículo [8] ha obtenido un rendimiento mucho mejor que `intlinprog` sobre la colección NGCUT usada, sobre todo si se tiene en cuenta que el algoritmo del artículo fue ejecutado en un ordenador de 1998, mientras que el algoritmo de `intlinprog` ha sido ejecutado en uno actual. Pero, ¿esto permite asegurar que el algoritmo `intlinprog` es peor que el del artículo [8]? Resulta que el algoritmo que se utiliza en [8] ha sido especialmente diseñado para resolver problemas 2DBPP, mientras que en `intlinprog` de Matlab es un software generalista de programación entera mixta. Es evidente que la comparación no ha sido justa. Para intentar equilibrar la balanza, en la ejecución de `intlinprog` hemos añadido dos aspectos clave del artículo en cuestión, que son la utilización de algoritmos heurísticos y la información dada por las cotas inferiores sobre el número necesario de contenedores para el empaquetamiento (ver Tabla 5.3). El objetivo de utilizar un algoritmo heurístico es obtener una distribución de los rectángulos en los distintos contenedores, que posiblemente no sea la óptima en el sentido de que utilice el menor número de contenedores posibles, pero que verifica las restricciones del problema. Esto va a permitir obtener un punto inicial admisible para que sea utilizado

por `intlinprog`, lo que en muchos casos suele mejorar el rendimiento de dicho algoritmo. En cualquier caso, la solución obtenida por el algoritmo heurístico aporta una cota superior de la solución, que es de gran utilidad ya que como también se dispone de una cota inferior, en el caso de alcanzarse la igualdad, la solución alcanzada por el algoritmo heurístico es también solución de 2DBPP, sin necesidad de ejecutarse tan siquiera el algoritmo `intlinprog`. Por último, se pueden utilizar la cota superior e inferior calculadas como restricciones de la variable v.

El algoritmo heurístico elegido en este caso es Finite Next-Fit, FNF, descrito por primera vez en [3] y cuyo pseudocódigo se muestra en el Algoritmo 6. Es un algoritmo que coloca los distintos rectángulos por niveles en los distintos contenedores. Para ello los rectángulos tienen que estar ordenados de mayor a menor altura. De acuerdo con este orden, los rectángulos son colocados por niveles en el contenedor, empezando por la parte inferior izquierda del mismo. Cuando se quiere colocar un nuevo rectángulo, primero se intenta colocar en el nivel abierto. Si no hay suficiente espacio se cierra el nivel y se crea uno nuevo en el contenedor. Si aún así no hay suficiente espacio para colocar el rectángulo, el contenedor se cierra y se añade un nuevo contenedor donde el rectángulo es colocado. De esta forma, solamente hay un contenedor abierto y cuando se cierra un contenedor no se vuelve a abrir. La ventaja principal de este algoritmo es su coste temporal lineal $O(n)$. Como aspecto negativo es que no ofrece la mejor cota superior, aunque para el conjunto de problemas NGCUT escogidos en los que el número de rectángulos involucrados es bajo, no se observa mucha diferencia con otros algoritmos heurísticos mejores en este aspecto como Finite First-Fit, FFF, o Finite Best-Strip, FBS, que son utilizados en [8] y fueron propuestos originalmente en [3], cuyo coste temporal es mayor: cuadrático $O(n^2)$ y linealítmico $O(n \log n)$ respectivamente.

Los resultados tras aplicar esas dos mejoras al algoritmo original de `intlinprog` se pueden observar en la Tabla 5.3, donde se muestran también las cotas superiores obtenidas con el algoritmo FNF (ver columna etiquetada con UB).

Problema	Tiempo intlinprog /s	N_C intlinprog	C_M	UB
NGCUT1	714.32	3	2	3
NGCUT2	2000	4	3	4
NGCUT3	2000	4	3	4
NGCUT4	0	2	–	2
NGCUT5	0	3	3	4
NGCUT6	2000	3	2	4
NGCUT7	0.12	1	1	2
NGCUT8	20.65	2	2	3
NGCUT9	2000	3	3	4
NGCUT10	0	3	–	3
NGCUT11	2000	3	2	3
NGCUT12	2000	4	3	4

Tabla 5.3: Resultados de la ejecución de `intlinprog` para las 12 instancias NGCUT (con punto inicial).

En esta tabla se puede observar que se resuelven seis de las doce problemas analizados, mejorando notablemente los resultados de la tabla anterior. De esos seis casos, dos de ellos, NGCUT 4 y NGCUT 10, se resolvieron previamente a la llamada al código `intlinprog`, gracias al FNF y las cotas inferiores. Se puede apreciar también que en la columna tercera se han resaltado en rojo tres de las instancias para las que la solución obtenida en 2000 s utiliza un contenedor más que la solución óptima de [8]. Por último, la cota inferior

calculada por `intlinprog`, C_M, y que coincide en todos los casos con la cota inferior L_4 pasada como parámetro, mejora en varios casos respecto a la Tabla 5.2.

Algoritmo 6: Finite Next-Fit(W, H, w, h)

Entrada: W: anchura del contenedor.
H: altura del contenedor.
w: array de las anchuras de los rectángulos ordenados de mayor a menor altura.
h: array de las alturas de los rectángulos ordenados de mayor a menor altura.

Salida : Vector (x, y, m) formado por las coordenadas de las esquinas inferior izquierda de cada rectángulo y el número de contenedor al que pertenecen.

```
/* Inicialización variables auxiliares                    */
```
$numContenedor = 1$; $anchuraRestante = W$; $alturaRestante = H$;
$alturaNivel = 0$;
```
/* Inicialización variables de salida.                    */
```
$x = [\,]$; $y = [\,]$; $m = [\,]$;
```
/* Empiezan las iteraciones.                              */
```
$alturaRestante = alturaRestante - h(1)$;
para $i \leftarrow 1$ **a** *longitud(w)* **hacer**
  si $anchuraRestante < w(i)$ **entonces**
    $anchuraRestante = W$;
    si $alturaRestante >= h(i)$ **entonces**
      $alturaNivel = H - alturaRestante$;
      $alturaRestante = alturaRestante - h(i)$;
    en otro caso
      $numContenedor = numContenedor + 1$;
      $alturaRestante = H - h(i)$;
      $alturaNivel = 0$;
    fin
  fin
  $x(i) = W - anchuraRestante$;
  $y(i) = alturaNivel$;
  $m(i) = numContenedor$;
  $anchuraRestante = anchuraRestante - w(i)$;
fin
devolver (x, y, m)

5.2. Resolviendo el modelo Set-Covering

En esta sección se pretende mostrar los resultados obtenidos tras resolver el Master Problem (MP) mediante la resolución iterativa del Restricted Master Problem (RMP) y del Pricing Problem (PP) tal como ha sido explicado en los capítulos 3 y 4.

En primer lugar y a modo de ejemplo se procede a resolver el problema NGCUT1 de

forma detallada:

Ejemplo 5.1. *Sea el problema NGCUT1 en el que se precisa colocar 10 rectángulos en el mínimo número de contenedores de dimensiones* $W = H = 10$. *Los rectángulos se encuentran ordenados por altura decreciente y sus anchuras y alturas vienen dadas por los siguientes vectores:*

$$w = (2, 2, 2, 4, 4, 4, 7, 7, 9, 9), \;\; h = (10, 8, 8, 5, 5, 5, 3, 3, 2, 2).$$

Resolver el (MP) de forma iterativa utilizando el algoritmo 1.

En primer lugar hay que ejecutar la heurística FNF mostrada en el algoritmo 6, para la cual se precisa tener los rectángulos ordenados por alturas en orden decreciente. El resultado, que se puede ver en la Figura 5.1, muestra que los rectángulos pueden ser colocados en tres contenedores, lo que supone una cota superior al problema ($UB = 3$) y es además un punto admisible del mismo. Las cotas inferiores son $L_0 = L_4 = 2$, por lo tanto, no se da la igualdad entre cotas superior e inferior y no se puede asegurar que el punto admisible dado por la heurística FNF sea una solución del problema.

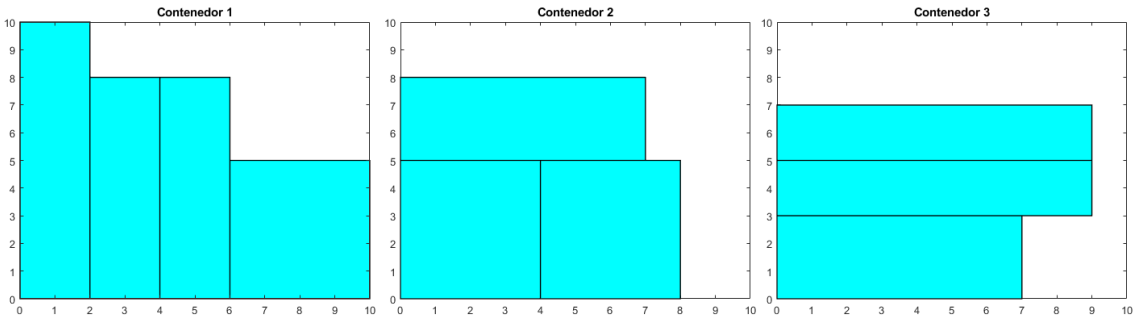


Figura 5.1: Resultado dado por la heurística FNF.

El siguiente paso es ejecutar el método de generación de columnas para intentar mejorar el resultado por el algoritmo FNF o bien intentar asegurar que el valor óptimo de la función objetivo es 3. Para la ejecución del algoritmo 1 se precisa disponer los rectángulos ordenados por área decreciente, con el fin de lograr una ejecución eficiente del algoritmo MAC. Los vectores w y h de acuerdo al nuevo orden se muestran a continuación:

$$w = (7, 7, 2, 4, 4, 4, 9, 9, 2, 2), \;\; h = (3, 3, 10, 5, 5, 5, 2, 2, 8, 8).$$

De esta forma, ahora se inicializan las matrices de patrones, S, y las de coordenadas, x e y, con el resultado dado por FNF, de acuerdo a la ordenación por área explicada previamente. Nótese que el símbolo $*$ que aparece en las matrices de las coordenadas corresponden a rectángulos que no están en el patrón y por lo tanto no se les ha asignado valor alguno.

$$S = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix};\ x = \begin{pmatrix} * & 0 & * \\ * & * & 0 \\ 0 & * & * \\ 6 & * & * \\ * & 0 & * \\ * & 4 & * \\ * & * & 0 \\ * & * & 0 \\ 2 & * & * \\ 4 & * & * \end{pmatrix}; y = \begin{pmatrix} * & 5 & * \\ * & * & 0 \\ 0 & * & * \\ 0 & * & * \\ * & 0 & * \\ * & 0 & * \\ * & * & 3 \\ * & * & 5 \\ 0 & * & * \\ 0 & * & * \end{pmatrix}$$

A continuación, se resuelve el primer problema (RMP) utilizando `linprog` con el fin de obtener el vector de los multiplicadores de Lagrange asociados a las restricciones generales, $\hat{\mu}$, y la solución, $\bar{x}$:

$$\hat{\mu} = (0,0,0,0,0,1,0,1,0,1)^T,\ \bar{x} = (1,1,1)^T,\ f(\bar{x}) = 3.$$

Ahora con el vector de multiplicadores de Lagrange y con los datos de los rectángulos y del contenedor, se procede a resolver el *Pricing Problem* de forma iterativa en dos etapas. La primera corresponde a la resolución del (1DMCKP), donde inicialmente el conjunto $\mathcal{C}$ es vacío. Recordemos que este conjunto contiene información sobre aquellos conjuntos de rectángulos que no pueden ser empaquetados en un solo contenedor. Por lo tanto, se resuelve el siguiente problema:

$$\text{(1DKPP)} \begin{cases} \text{Maximizar} & f(u) = u_6 + u_8 + u_{10} \\ u \in \{0,1\}^{10} & \\ \text{Sujeto a:} & \sum_{i=1}^{10} w_i h_i u_i \leq 100. \end{cases}$$

La solución del problema obtenida por `intlinprog` es:

$$\bar{u} = (0,0,0,0,0,1,0,1,0,1)^T,\ f(\bar{u}) = 3.$$

Entonces la segunda parte corresponde a comprobar si los rectángulos 6, 8 y 10 pueden ser empaquetados en un solo contenedor. Para ello se dispone del algoritmo MAC explicado en el capítulo 4, cuya ejecución en este caso determina que esos rectángulos caben en un solo contenedor y, por lo tanto, se ha resuelto el *Pricing Problem* determinándose la nueva columna para añadir a la matriz S, $\bar{u}$. Además, el algoritmo MAC proporciona las correspondientes nuevas columnas a añadir a las matrices x e y que vienen dadas respectivamente por los vectores:

$$(*,*,*,*,*,0,*,0,*,4)^T \text{ y } (*,*,*,*,*,0,*,8,*,0)^T.$$

En este punto hay que comprobar si la solución del (RMP), $\bar{x} = (1,1,1)^T$, da lugar a una solución del (MP), para ello se comprueba si se verifica la desigualdad de la hipótesis del Teorema 3.1. Como $1 - \bar{u}^T = -1$, la desigualdad no se verifica, por lo tanto, hay

que añadir el patrón $\bar{u}$ obtenido como solución del *Pricing Problem* a la matriz S y las coordenadas asociadas a los rectángulos del nuevo patrón. A continuación se resuelve otro (RMP) y el (PP) asociado hasta obtener la solución del (MP). Debido al elevado número de iteraciones necesarias para ello, se procede a presentar la solución del (MP) junto con otros datos de ejecución del algoritmo 1. A continuación se muestran la solución del (MP), $\bar{x}^{(MP)}$, donde se alcanza un valor de la función objetivo de $f(\bar{x}^{(MP)}) = 2.2$, junto con las matrices S, x y y finales.

$$\bar{x}^{(MP)} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0, 0, 0.1, 0.1, 0, 0.1, 0.7, 0.1, 0.3, 0, 0, 0.2, 0.5)$$

$$S = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1
\end{pmatrix}$$

$$x = \begin{pmatrix}
* & 0 & * & * & * & * & 0 & 0 & * & * & 0 & 0 & * & 0 & * & 0 & 0 & 0 & 0 & * & * & * & * & * & * \\
* & * & 0 & * & * & * & * & * & * & 0 & * & 0 & 0 & 0 & 0 & * & * & * & 0 & * & * & 0 & 0 & 0 & * \\
0 & * & * & * & * & * & * & 8 & * & * & * & * & * & * & * & * & 8 & * & * & * & 0 & * & * & 8 & 0 \\
6 & * & * & * & * & * & 0 & * & 0 & 0 & 0 & * & * & * & 0 & * & 0 & * & * & 0 & * & 0 & 0 & 0 & 2 \\
* & 0 & * & * & * & 0 & * & 0 & 4 & * & * & * & 0 & * & * & 0 & 4 & 0 & * & * & 2 & 4 & 4 & * & 2 \\
* & 4 & * & 0 & 0 & * & * & 4 & * & 4 & 4 & * & 4 & * & 4 & 4 & * & 4 & * & 4 & 2 & * & * & 4 & * \\
* & * & 0 & * & 0 & * & * & * & 0 & * & * & * & * & 0 & 0 & * & * & 0 & 0 & 0 & * & * & 0 & * & * \\
* & * & 0 & 0 & * & 0 & 0 & * & * & * & 0 & * & 0 & * & * & 0 & * & * & 0 & 0 & * & 0 & * & * & * \\
2 & * & * & * & 4 & 4 & * & * & * & * & 8 & * & 8 & 7 & * & * & * & 8 & * & * & 6 & * & 8 & * & 6 \\
4 & * & * & 4 & * & * & * & * & 8 & * & * & 7 & * & * & 8 & 8 & * & * & * & * & 8 & 8 & * & * & 8
\end{pmatrix}$$

$$y = \begin{pmatrix}
* & 5 & * & * & * & * & 0 & 0 & * & * & 0 & 0 & * & 0 & * & 0 & 0 & 0 & 0 & * & * & * & * & * & * \\
* & * & 0 & * & * & * & * & * & * & 0 & * & 3 & 0 & 3 & 0 & * & * & * & 3 & * & * & 0 & 0 & 0 & * \\
0 & * & * & * & * & * & * & 0 & * & * & * & * & * & * & * & * & 0 & * & * & * & 0 & * & * & 0 & 0 \\
0 & * & * & * & * & * & 3 & * & 0 & 3 & 3 & * & * & * & 3 & * & 3 & * & * & 0 & * & 3 & 3 & 3 & 0 \\
* & 0 & * & * & * & 0 & * & 3 & 0 & * & * & * & 3 & * & * & 3 & 3 & 3 & * & * & 0 & 3 & 3 & * & 5 \\
* & 0 & * & 0 & 0 & * & * & 3 & * & 3 & 3 & * & 3 & * & 3 & 3 & * & 3 & * & 0 & 5 & * & * & 3 & * \\
* & * & 3 & * & 8 & * & * & * & 8 & * & * & * & * & 8 & 8 & * & * & 8 & 6 & 5 & * & * & 8 & * & * \\
* & * & 5 & 8 & * & 8 & 8 & * & * & * & 8 & * & 8 & * & * & 8 & * & * & 8 & 7 & * & 8 & * & * & * \\
0 & * & * & * & 0 & 0 & * & * & * & * & 0 & * & 0 & 0 & * & * & * & 0 & * & * & 0 & * & 0 & * & 0 \\
0 & * & * & 0 & * & * & * & * & 0 & * & * & 0 & * & * & 0 & 0 & * & * & * & * & 0 & 0 & * & * & 0
\end{pmatrix}$$

El valor $\lceil f(\bar{x}^{(MP)}) \rceil = \lceil 2.2 \rceil = 3$ es una cota inferior del número de contenedores necesarios para el empaquetamiento de los rectángulos del problema NGCUT1 ya que

debido a las características de la función objetivo del modelo (3.1) los únicos valores que puede tomar ésta son enteros y, por tanto, para obtener una cota inferior a partir de la solución del (MP) hay que aplicar la función techo. En este caso, ese valor mejora las cotas L_0 y L_4 (ambas iguales a 2) y confirma que la solución dada por la heurística FNF (ver Figura 5.1) es la solución óptima del problema (2DBPP) ya que la cota inferior y la superior coinciden.

La ejecución completa del código ha tardado 2.18 s, por lo que ha sido muy rápida, obteniendo una matriz S con un total de 25 columnas, de las cuales 22 han sido generadas propiamente en el proceso iterativo de resolución del (PP) y alcanzando las 181 restricciones generales para el último (1DMCKP) resuelto. Además, se han resuelto 203 problemas auxiliares del tipo (1DMCKP). En la Figura 5.2 se muestran cuatro gráficas con resultados asociados a la ejecución del algoritmo de generación de columnas: la evolución del valor de la función objetivo de los problemas del tipo (RMP) y de los del tipo (PP), la evolución del número de restricciones generales de los problemas (1DMCKP) (igual al número de filas de A_{KP}) y el número de nodos explorados por el algoritmo Branch and Bound para la resolución de cada (1DMCKP), de arriba a abajo y de izquierda a derecha respectivamente. Como se puede apreciar en dicha figura, al aumentar el número de filas de la matriz A_{KP} tiende a aumentar el número de nodos explorados por el algoritmo `intlinprog`.

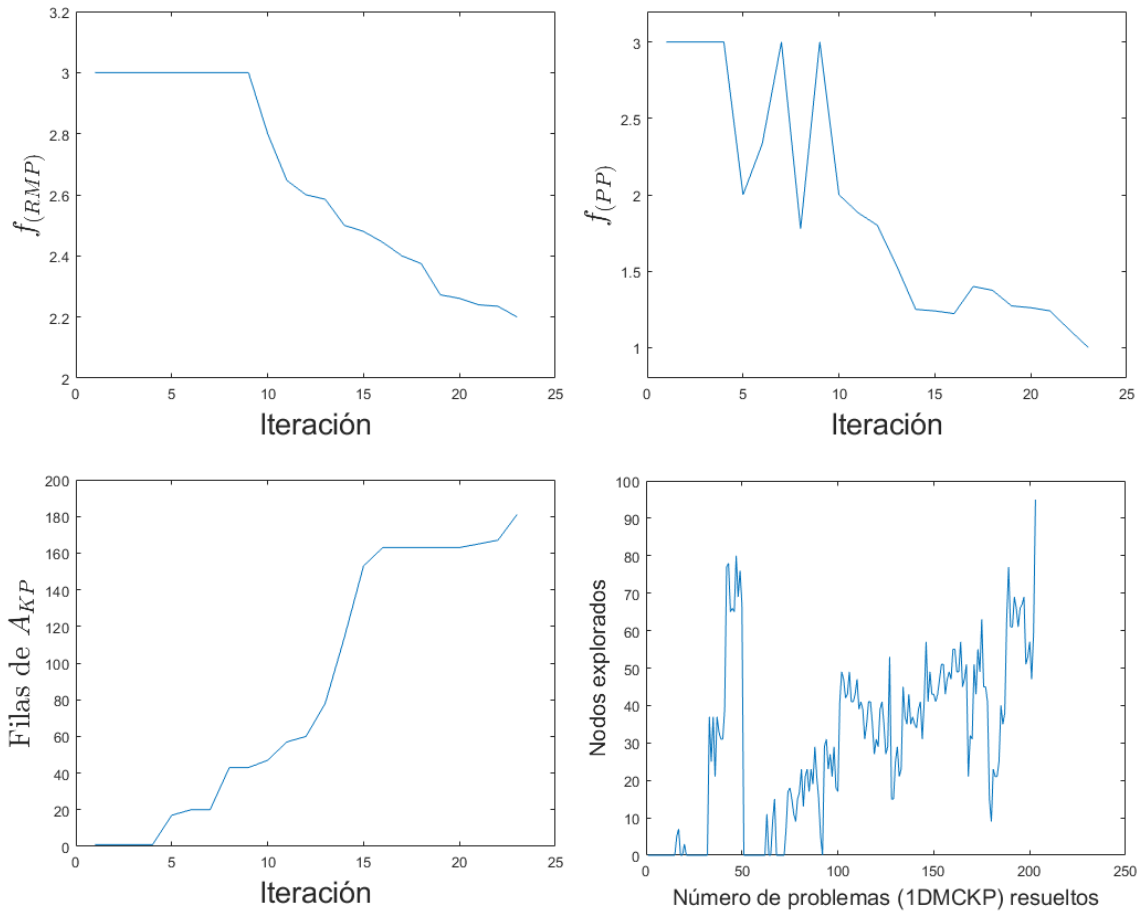


Figura 5.2: Resultados con el algoritmo de generación de columnas para el problema NGCUT1.

En este ejemplo se ponen de manifiesto dos aspectos a tener en cuenta en la resolución numérica de (2DBPP). En primer lugar, tal como se expone en [10] el número de restricciones de la matriz A_{KP} llega a ser muy grande, haciendo más difícil a `intlinprog` resolver los (1DMCKP) ya que cada vez tiene que explorar una mayor cantidad de nodos. Este aspecto se hace más patente en problemas con un mayor número de rectángulos, llegando a la situación en la que `intlinprog` pierde viabilidad y es abortada la ejecución enviando un mensaje de problema mal condicionado. En segundo lugar, se puede concluir que resolver el Pricing Problem es muy caro. De hecho, en [10] evitan resolverlo de forma exacta siempre y cuando sea posible y hacen uso de un algoritmo heurístico para la resolución aproximada del mismo. Según datos del mismo artículo, la resolución exacta del (PP) ocupa más del 90 % del tiempo de CPU de ejecución del algoritmo de generación de columnas.

En este proyecto el algoritmo heurístico no ha sido programado y los esfuerzos se han centrado en la matriz de restricciones generales A_{KP}. Resulta que las restricciones que se van añadiendo de la forma (4.2) son débiles, por ello, uno de los aspectos en los que nos vamos a centrar es cómo conseguir restricciones generales más fuertes en el sentido que excluyen un mayor número de patrones. La respuesta pasa por obtener el conjunto mínimo no admisible (MIS, del inglés *Minimal Infeasible Set*), que consiste en el conjunto más pequeño de rectángulos que no pueden ser colocados en un solo contenedor, pero que al eliminar cualquiera de ellos el conjunto pasa a ser admisible. Esto se ha abordado con un algoritmo MIS heurístico (algoritmo voraz) y con un algoritmo MIS exacto. En el caso del algoritmo voraz, este consistía en eliminar los rectángulos de menor área hasta obtener un conjunto admisible de rectángulos. Entonces dicho conjunto con el último rectángulo eliminado conforman la solución heurística del algoritmo voraz. En cambio en el algoritmo exacto se comprueban las combinaciones de dos rectángulos en adelante hasta obtener un conjunto de rectángulos que no pueden ser colocados en un único contenedor. En ese caso se terminan de analizar todas las combinaciones de ese mismo número de rectángulos con el fin de obtener todos los MIS existentes. El algoritmo MIS exacto es muy costoso en general, pero experimentos numéricos sobre la colección de problemas NGCUT han mostrado que su utilización resulta muy beneficiosa, mejorando mucho el rendimiento en comparación con el algoritmo MIS heurístico.

En la Tabla 5.4 se presentan los resultados numéricos obtenidos con las últimas mejoras comentadas en la resolución de los 12 problemas de la colección NGCUT. Para ello se ha hecho uso del algoritmo MIS exacto para introducir restricciones más fuertes a las matrices A_{KP}. Además, para obtener una solución entera se ha tratado de abordar el salto entre el (MP) (programación lineal continua) y el (SCM) (programación entera mixta) resolviendo un problema reducido asociado a la última matriz S calculada dada por el último (RMP). Notemos que gracias a que en la ejecución usamos el algoritmo FNF, la existencia de puntos admisibles está garantizada al igual que la existencia de solución para el problema entero reducido. Salvo coincidencia del valor de la función objetivo con la cota inferior, no podrá asegurarse que la solución del (SCM) reducido proporciona una solución óptima del (SCM).

En concreto en la tabla 5.4 se muestran en columnas para cada problema el tiempo necesario para resolver el (MP), el correspondiente valor de la función objetivo, el número de patrones usados, el número de restricciones en el último problema (1DMCKP), el número de problemas (1DMCKP) resueltos, el número de contenedores propuestos para

el empaquetamiento y el valor de la cota inferior L_4 y la cota superior UB.

De la tabla siguiente hay varios aspectos que comentar. En primer lugar, se han conseguido resolver un total de 11 (MP) (todos excepto el NGCUT3 porque se alcanzó la limitación de 2000 s de tiempo de ejecución) y de esos problemas se han conseguido resolver hasta la optimalidad un total de 6 instancias NGCUT. La optimalidad queda garantizada cuando se produce la igualdad entre la cota inferior L_4 o la dada por la solución del (MP) y el valor de la columna (SCM), o bien porque la solución del (MP) es entera. Aquellos problemas donde no se ha garantizado la optimalidad han sido marcados con color rojo claro. Nótese que en dos de ellos, el NGCUT2 y el NGCUT6 se ha alcanzado el óptimo aunque el código no lo ha detectado. Por otro lado, la relajación continua LP realizada en el (MP) permite obtener una cota inferior que mejora en el NGCUT1 y NGCUT6 la cota L_4 e iguala en los restantes problemas NGCUT. Por último, una comparación del ejemplo 5.2 con los valores mostrados para el problema NGCUT1 en la Tabla 5.4 permiten comprobar que tanto el tiempo de ejecución del algoritmo, como el número de columnas de S, el número de filas de A y el número de (1DMCKP) resueltos se ha visto reducido considerablemente (siendo antes los valores 2.18 s, 25 columnas, 181 filas y 203 problemas, respectivamente).

Problema	Tiempo (MP) / s	$f(\bar{x}^{(MP)})$	Columnas de S	Filas de A	Número de (1DMCKP) resueltos	(SCM)	L_4	UB
NGCUT1	0.74	2.20	21	22	36	3	2	3
NGCUT2	2.20	3.00	46	152	161	4	3	4
NGCUT3	>2000.00						3	4
NGCUT4	0.66	1.25	26	2	26	2	2	2
NGCUT5	12.30	3.00	23	540	337	4	3	4
NGCUT6	6.34	2.14	35	317	250	3	2	4
NGCUT7	0.54	1.00	17	1	16	1	1	2
NGCUT8	12.19	1.77	36	221	184	2	2	3
NGCUT9	7.28	2.66	41	378	311	3	3	4
NGCUT10	1.62	2.10	48	77	121	3	3	3
NGCUT11	448.68	2.00	39	136	98	3	2	3
NGCUT12	72.75	2.83	43	1022	729	4	3	4

Tabla 5.4: Resultados para la colección de problemas NGCUT utilizando el algoritmo 1 completo.

¿Y qué ocurre con aquellos problemas para los que no hemos detectado la optimalidad dada por [8] (en nuestro los NGCUT 2, 3, 5, 6, 11 y 12)? Pues bien, aunque esta parte no se va a tratar en este TFG, lo que hace el artículo [10] es utilizar una técnica de Branch and Bound utilizando la estrategia de ramificación de Ryan-Foster. Básicamente consiste en seleccionar dos rectángulos del patrón o columna de S con el valor fraccionario más elevado y dividir el espacio de soluciones en dos ramas: una en la que los dos rectángulos están en el mismo contenedor y otra en la que los dos rectángulos se encuentran en contenedores distintos. De esta forma estamos tomando dos rectángulos que probablemente estén en el mismo contenedor en una solución óptima.

El algoritmo de generación de columnas que se ha presentado en este proyecto es el actual estado del arte en la resolución del (2DBPP) [5] y supone una mejora importante en comparación con el algoritmo de [8] en la resolución de problemas con una distribución parecida de rectángulos pequeños, medianos y grandes. Este rendimiento tan bueno que se muestra en [10] no se ha podido alcanzar en este proyecto entre otros motivos por el lenguaje de programación empleado (Matlab en vez de C), si hablamos de tiempo de ejecución, los programas usados (en algunos casos no especializados a los problemas a

resolver), así como la ausencia de un algoritmo heurístico para la resolución del (PP). Pese a eso, el trabajo efectuado a lo largo de este proyecto ha permitido estudiar en detalle, tanto de forma teórica como práctica, las formulaciones de los distintos problemas y los principales algoritmos utilizados en el método de generación de columnas para la resolución del (2DBPP), así como comprobar algunas de las afirmaciones realizadas en [10].

Bibliografía

[1] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Oper. Res. Lett.*, 33(1):42–54, Jan. 2005.

[2] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[3] J. O. Berkey and P. Y. Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38(5):423–429, 1987.

[4] Intlinprog. Algoritmos de programación lineal entera mixta. `https://es.mathworks.com/help/optim/ug/mixed-integer-linear-programming-algorithms.html`. Consultado por última vez el 25/08/2021.

[5] M. Iori, V. L. de Lima, S. Martello, F. K. Miyazawa, and M. Monaci. Exact solution techniques for two-dimensional cutting and packing. *European Journal of Operational Research*, 289(2):399–415, 2021.

[6] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[7] S. Martello, D. Pisinger, D. Vigo, E. Boef, and J. Korst. Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Softw.*, 33:7, 01 2007.

[8] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44(3):388–399, 1998.

[9] NGCUT. Repositorio. `http://or.dei.unibo.it/library/2dpacklib`. Consultado por última vez el 17/02/2021.

[10] D. Pisinger and M. Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, 19(1):36–51, 2007.

[11] S. J. Russell and P. Norvig. *Inteligencia artificial. Un enfoque moderno.* Pearson, 2004.

[12] L. A. Wolsey. *Integer Programming.* Wiley, 1998.

Glosario

A continuación se introducen algunas de las notaciones utilizadas en el trabajo:

- Dado $x \in \mathbb{R}$, la función techo viene dada por $\lceil x \rceil = \min\{k \in \mathbb{Z} : x \leq k\}$.
- Dado $x \in \mathbb{R}$, la función suelo viene dada por $\lfloor x \rfloor = \max\{k \in \mathbb{Z} : k \leq x\}$.
- Dados $x, y \in \mathbb{R}^n$, $x \leq y$ significa que $x_i \leq y_i$ para $i = 1, \ldots, n$.
- Dada un conjunto A, denotamos con $|A|$ el cardinal del conjunto A.
- 1DKPP: Problema Unidimensional de la Mochila.
- 1DMCKP: Problema Unidimensional de la Mochila con Múltiples Restricciones.
- 2DBPP: Problema bidimensional de empaquetamiento.
- CSP: Problema de Satisfacción de Restricciones.
- IP: Programación Entera.
- LP: Programación Lineal Continua.
- MAC: algoritmo de Búsqueda Manteniendo Arco-Consistencia.
- MILP: Programación Lineal Entera Mixta.
- MIP: Programación Entera Mixta.
- MIS: conjunto mínimo no admisible.
- MP: Master Problem.
- PP: Pricing Problem.
- RMP: Restricted Master Problem.
- SCM: Modelo de Cobertura de Conjuntos.

Apéndice A

Contenidos básicos de optimización

Sea el problema de programación lineal (PL) que se muestra a continuación:

$$(PL)\begin{cases}\text{Minimizar } f(x)=c^Tx\\ x\in\mathbb{R}^n\\ \text{Sujeto a:} \qquad a_i^Tx=b_i,\ i=1,\dots,n_I\\ \qquad\qquad\quad \widetilde{a_j}^Tx\le\widetilde{b_j},\ j=1,\dots,n_D,\end{cases} \tag{A.1}$$

donde $c, a_i, \widetilde{a_j}\in\mathbb{R}^n$ y $b_i, \widetilde{b_j}\in\mathbb{R}$.

Definición A.1. *Se llama* ***función objetivo*** *del problema (PL) a la función lineal c^Tx que se quiere minimizar.*

Definición A.2. *Las restricciones de un problema de programación lineal se pueden clasificar en dos grupos:* ***restricciones de cota*** *y* ***restricciones generales****. Las primeras afectan a cada variable de forma individual y las segundas afectan a dos o más variables.*

Definición A.3. *El* ***conjunto de puntos admisibles*** *de (PL), denotado por $\mathcal{A}$, está formado por aquellos elementos de $\mathbb{R}^n$ que verifican las restricciones del problema (PL).*

Teorema A.4. *(Existencia de solución) Si el conjunto $\mathcal{A}$ es no vacío y el $\inf\{c^Tx : x\in\mathcal{A}\}$ pertenece a $\mathbb{R}$, entonces existe solución global para el problema de optimización (PL).*

A continuación se enuncian las condiciones de optimalidad de primer orden.

Teorema A.5. *(Condición de optimalidad necesaria y suficiente). $\bar{x}$ es solución global del problema (PL) si y solo si existen: $\{\bar{\lambda}_i\}_{i=1}^{n_I}\subset\mathbb{R}$ y $\{\bar{\mu}_j\}_{j=1}^{n_D}\subset\mathbb{R}_+$ tales que*

$$\bar{x} \text{ es admisible,} \tag{A.2}$$

$$c+\sum_{i=1}^{n_I}\bar{\lambda}_ia_i+\sum_{j=1}^{n_D}\bar{\mu}_j\widetilde{a_j}=0, \tag{A.3}$$

$$\bar{\mu}_j\ge 0 \text{ y } \bar{\mu}_j(\widetilde{a_j}^T\bar{x}-\widetilde{b_j})=0,\ j=1,\dots,n_D, \tag{A.4}$$

Definición A.6. *A los números $\{\bar{\lambda}_i\}_{i=1}^{n_I}$ y $\{\bar{\mu}_j\}_{j=1}^{n_D}$ se les llama* ***multiplicadores de Lagrange****.*